PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (V)

Comunicația inter-procese:

Canale de comunicație anonime și cu nume

Cristian Vidrașcu
vidrascu@info.uaic.ro

Mai, 2020

Sumar

Introducere

Tipuri de comunicație între procese:

- comunicația prin memorie partajată (“*shared-memory communication*”)
  e.g. prin fișiere mapate în memorie, sau mapări anonime și cu nume, ș.a.

- comunicația prin schimb de mesaje (“*message-passing communication*”)
  - *comunicație locală*
    - canale anonime (numite, uneori, și canale interne)
    - canale cu nume, *i.e.* fișiere fifo (numite, uneori, și canale externe)
  - *comunicație la distanță*
    - *socket*-uri

Introducere (cont.)

Un *canal de comunicație* UNIX, sau *pipe*, este o "conductă" prin care pe la un capăt se scriu mesajele (ce constau în secvențe de octeți), iar pe la celălalt capăt acestea sunt citite (cu extracția lor din canal) – deci practic se comportă ca o structură de tip coadă, adică o listă FIFO (*First-In,First-Out*).
Notă: de fapt, un *pipe* chiar este implementat de nucleul UNIX / Linux ca o listă FIFO, cu o capacitate constantă, gestionată în *kernel-space*.

Rolul unui canal: o asemenea "conductă" FIFO poate fi folosită pentru comunicare de către două (sau mai multe) procese, pentru a transmite date de la unul la altul (!).

Canalele de comunicație UNIX se împart în două subcategorii:

- *canale anonime*: aceste "conducte" sunt create în memoria internă a sistemului UNIX respectiv, fără niciun nume asociat lor în sistemul de fișiere;
- *canale cu nume*: aceste "conducte" sunt create tot în memoria internă a sistemului, dar au asociate câte un nume, reprezentat printr-un fișier de tipul special *fifo*, care este păstrat în sistemul de fișiere (din acest motiv, aceste fișiere *fifo* se mai numesc și *pipe*-uri cu nume).

Canale anonime

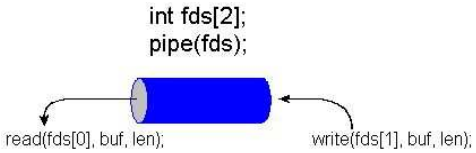
Agenda

Crearea lor, cu primitiva `pipe()`

Un *canal anonim* se creează cu ajutorul primitivei `pipe`.
Interfața acestei funcții este următoarea ([3]):
`int pipe(int *p)`

- *p* = parametrul efectiv de apel trebuie să fie un vector `int[2]`, care va fi actualizat de funcție în felul următor:
  - *p*`[0]` va fi descriptorul de fișier deschis pentru *capătul de citire* al canalului
  - *p*`[1]` va fi descriptorul de fișier deschis pentru *capătul de scriere* al canalului
- valoarea returnată este `0`, în caz de succes, sau `-1`, în caz de eroare.

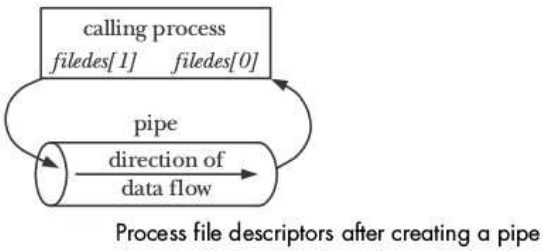
Efect: în urma execuției primitivei `pipe` se creează un canal anonim și *este deschis automat la ambele capete* – în citire la capătul referit prin descriptorul *p*`[0]` și, respectiv, în scriere la capătul referit prin descriptorul *p*`[1]`.



6 / 41

Modul de utilizare a unui canal anonim

După crearea unui canal anonim, folosirea sa pentru comunicația locală între două (sau mai multe) procese se face prin scrierea informației în acest canal și, respectiv, prin citirea informației din canal.



Process file descriptors after creating a pipe

Iar scrierea în canal și respectiv citirea din canal, prin intermediul celor doi descriptori *p*`[0]` și *p*`[1]`, se efectuează la fel ca pentru fișierele obișnuite, *i.e.* folosind apelurile `read` și `write`, sau cu funcțiile I/O din biblioteca `stdio`.

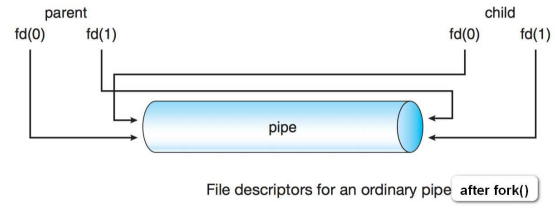
Restricție importantă:
Deoarece acest tip de canale sunt *anonime* (*i.e.*, nu au nume), pot fi utilizate pentru comunicație doar de către procese "înrudite" prin apeluri `fork`/`exec`.
De ce? Motivația este următoarea: ... (vezi slide-ul următor)

7 / 41

Modul de utilizare a unui canal anonim (cont.)

Motivație: pentru ca două (sau mai multe) procese să poată folosi un canal anonim pentru a comunica între ele, acele procese trebuie sa aibă la dispoziție cei doi descriptori *p*`[0]` și *p*`[1]` obținuți prin crearea canalului. Deci procesul care a creat canalul prin apelul `pipe`, va trebui să le "transmită" cumva celuilalt proces.

De exemplu, în cazul când se dorește să se utilizeze un canal anonim pentru comunicarea între două procese de tipul părinte-fiu, atunci este suficient să se apeleze primitiva `pipe` de creare a canalului *înaintea* apelului primitivei `fork` de creare a procesului fiu. În acest fel, prin clonare, avem la dispoziție și în procesul fiu cei doi descriptori necesari pentru comunicare prin intermediul acelui canal anonim.



Notă: "Transmiterea" descriptorilor canalului are loc și în cazul apelului primitivelor `exec` (deoarece descriptorii de fișiere deschise se moștenesc prin `exec`).

Modul de utilizare a unui canal anonim (cont.)

Altă restricție:
Dacă
un proces își închide vreunul dintre capetele unui canal anonim, atunci nu mai are nicio posibilitate de a redeschide ulterior acel capăt al canalului.

Figura 1 – un canal anonim, imediat după apelul pipe:

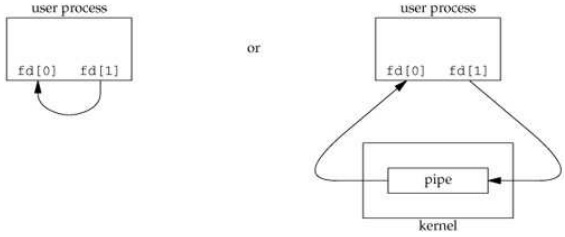


Figura 2 – un canal anonim, după apelul fork:

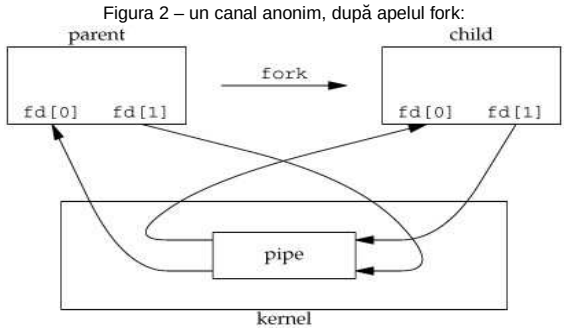
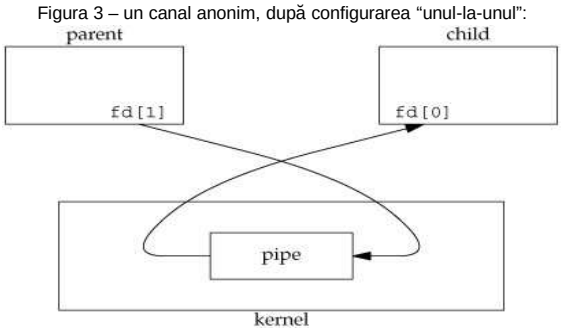


Figura 3 – un canal anonim, după configurarea "unul-la-unul":

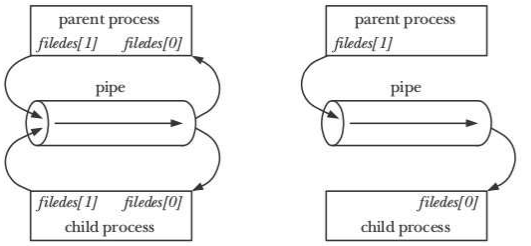


Notă: vom vedea ulterior că aceste două restricții de folosire a canalelor anonime nu mai sunt valabile și în cazul canalelor cu nume (!).

Demo: exemple de comunicație între două procese

- Primul exemplu: un program care exemplifică modul de utilizare a unui canal anonim pentru comunicația între două procese, de tipul producător–consumator.
  În acest exemplu se ilustrează folosirea primitivelor `read` și `write` (*i.e.*, funcțiile din API-ul `POSIX`) pentru a citi din canal, respectiv pentru a scrie în canal.

  A se vedea fișierul sursă `pipe_ex1.c` ([2]).



Setting up a pipe to transfer data from a parent to a child

Efectul acestui program: mai întâi, se creează un canal anonim și un proces fiu. Apoi, procesul părinte citește o secvență de caractere de la tastatură, secvență terminată cu combinația de taste `CTRL+D` (*i.e.*, caracterul `EOF` in `UNIX`), și le transmite procesului fiu, prin intermediul canalului anonim, doar pe acelea care sunt litere mici. Iar procesul fiu citește din canal caracterele trasmise de procesul părinte și le afișează pe ecran.

Notă: evident, se pot folosi și funcțiile I/O de nivel înalt (*i.e.*, cele din biblioteca standard I/O de C) pentru comunicația prin intermediul canalelor anonime.

10 / 41

Demo: exemple de comunicație între două procese (cont.)

- Primul exemplu: . . .

- Al doilea exemplu: un alt program care exemplifică folosirea unui canal anonim pentru comunicația între două procese, de tipul producător–consumator.
  De această dată, se utilizează funcțiile `fscanf` și, respectiv, `fprintf` (*i.e.*, din biblioteca `stdio`) pentru a citi din canal și, respectiv, pentru a scrie în canal.

  Notă: în acest caz, este necesară conversia descriptorilor de fișiere de la tipul `int` (*i.e.*, descriptorii folosiți de apelurile I/O din API-ul `POSIX`) la descriptori de tipul `FILE*` (*i.e.*, descriptorii folosiți de funcțiile I/O din biblioteca `stdio`), lucru realizabil cu ajutorul funcției de bibliotecă `fdopen`.

  A se vedea fișierul sursă `pipe_ex2.c` ([2]).

  Efectul acestui program: mai întâi, se creează un canal anonim și un proces fiu. Apoi, procesul tată citește o secvență de numere de la tastatură, secvență terminată cu combinația de taste `CTRL+D` (*i.e.*, caracterul `EOF` in `UNIX`), și le transmite procesului fiu, prin intermediul canalului anonim. Iar procesul fiu citește din canal numerele trasmise de procesul părinte și le afișează pe ecran.

11 / 41

Agenda

12 / 41

Crearea lor, cu primitiva `mkfifo()`

Un *canal cu nume* se creează cu ajutorul primitivei `mkfifo`.
Interfața acestei funcții este următoarea ([3]):
`int mkfifo(char *nume, int permisiuni)`

- *nume* = numele fișierului (de tip *fifo*) ce va fi creat
- *permisiuni* = permisiunile pentru fișierul ce va fi creat
- valoarea returnată este `0`, în caz de succes, sau `-1`, în caz de eroare.

Efect: în urma execuției primitivei `mkfifo` se creează un canal cu nume, dar *fără a fi deschis la ambele capete* (!), precum se întâmplă în cazul creării unui canal anonim.

Exemplu de creare a unui fișier *fifo*: a se vedea fișierul sursă `mkfifo_ex.c` ([2]).

Observație: crearea unui fișier *fifo* se mai poate face cu ajutorul primitivei `mknod` apelată cu *flag*-ul `S_IFIFO`. De asemenea, mai poate fi creat și direct de la linia de comandă (*i.e.*, prompterul *shell*-ului), cu comenzile `mkfifo` sau `mknod`.

13 / 41

Modul de utilizare a unui canal cu nume

După crearea unui canal cu nume, folosirea sa pentru comunicația locală între două (sau mai multe) procese se face prin scrierea informației în acest canal și, respectiv, prin citirea informației din canal.

Iar scrierea în canal și respectiv citirea din canal, se efectuează la fel ca pentru fișierele obișnuite, și anume: mai întâi se deschide *explicit* fișierul la "capătul" dorit (cel de citire și/sau cel de scriere), pentru a se obține descriptorul necesar, apoi se scrie în el și/sau se citește din el, prin intermediul descriptorului obținut explicit, i.e. folosind apelurile `read` și `write`, sau cu funcțiile de citire/scriere din biblioteca `stdio`, iar la sfârșit se închide descriptorul respectiv.

Observație importantă:
Deoarece acest tip de canale nu sunt *anonime* (*i.e.*, au nume prin care pot fi referite), pot fi utilizate pentru comunicație între orice procese care cunosc numele fișierului *fifo* respectiv, deci nu mai avem restricția de la canale anonime, aceea că procesele trebuiau să fie "înrudite" prin `fork`/`exec`.

14 / 41

Modul de utilizare a unui canal cu nume (cont.)

Așadar, operațiile asupra canalelor *fifo* se vor face fie cu primitivele I/O de nivel scăzut (*i.e.*, `open`, `read`, `write`, `close`), fie cu funcțiile I/O de nivel înalt din biblioteca standard de I/O din C (*i.e.*, `fopen`, `fread`/`fscanf`, `fwrite`/`fprintf`, `fclose`, ș.a.).

La fel ca pentru fișiere obișnuite, "deschiderea" unui fișier *fifo* se face explicit, printr-un apel al funcției `open` sau `fopen`, într-unul din următoarele trei moduri posibile, specificat prin parametrul transmis funcției de deschidere:

- *read* & *write* (*i.e.*, deschiderea ambelor capete ale canalului)
- *read-only* (*i.e.*, deschiderea doar a capătului de citire)
- *write-only* (*i.e.*, deschiderea doar a capătului de scriere)

Observație importantă:
Implicit, deschiderea se face în mod *blocant*, *i.e.* o deschidere *read-only* trebuie să se "sincronizeze" cu una *write-only*. Cu alte cuvinte, dacă un proces încearcă să deschidă un capăt al canalului, apelul funcției de deschidere rămâne blocat (*i.e.*, funcția nu returnează) până când un alt proces va deschide celălalt capăt al canalului.

15 / 41

Despre persistența informației dintr-un fișier *fifo*

Observație: un *canal cu nume* este creat tot în memoria internă a sistemului (ca și unul anonim), dar în plus are asociat un nume, reprezentat printr-un fișier de tipul special *fifo*, care este păstrat în sistemul de fișiere.

Concluzie: informațiile conținute în acest tip de fișiere sunt stocate în memoria principală, nu pe disc, și ca urmare nu sunt persistente. (Practic, conținutul unui fișier fifo este gestionat de SO tot ca o coadă FIFO aflată în memorie, la fel ca și în cazul canalelor anonime.)

Așadar, perioada de retenție a informației stocate într-un canal este următoarea:

Spre deosebire de fișierele obișnuite, care păstrează informația scrisă în ele pe perioadă nedeterminată (mai precis, până la o eventuală operație de modificare / ștergere), în cazul unui fișier *fifo* informația scrisă în canal se păstrează doar din momentul scrierii și până în momentul când atât procesul care a scris acea informație, cât și orice alt proces ce accesa acel canal, termină accesul la acel canal (închizându-și capetele canalului), iar aceasta numai dacă informația nu este consumată mai devreme, prin citire.
Demo: a se vedea fișierul sursă `testare_retentie_fifo.c` ([2]).

16 / 41

Deosebiri ale canalelor cu nume față de cele anonime

- Funcția de creare a unui fișier *fifo* (*i.e.*, canal cu nume) nu realizează și deschiderea automată a celor două capete ale canalului, precum la canalele anonime, ci acestea trebuie să fie deschise explicit, după creare, prin apelul unei funcții de deschidere a acelui fișier.
- Un canal *fifo* poate fi deschis, la oricare dintre capete, de orice proces, indiferent dacă acel proces are sau nu vreo legătură de "rudenie" (prin `fork` / `exec`) cu procesul care a creat canalul respectiv. Aceasta este posibil deoarece un proces trebuie doar să cunoască numele fișierului *fifo* pe care dorește să-l deschidă, pentru a-l putea deschide. Evident, mai trebuie și ca procesul respectiv să aibă drepturi de acces pentru acel fișier *fifo*.
- După ce un proces închide un capăt al unui canal *fifo*, acel proces poate redeschide din nou acel capăt al canalului.
  Motivul pentru care ar dori aceasta: poate constata, ulterior momentului închiderii, că are nevoie să mai efectueze și alte operații I/O asupra acelui capăt.

17 / 41

Agenda

Caracteristici și restricții ale canalelor de comunicație

- Ambele tipuri de canale sunt canale *unidirecționale*, adică pe la un capăt se scrie informația în canal, iar pe la capătul opus se citește.

  Notă: însă putem avea mai mulți scriitori (*i.e.*, toate procesele ce au acces la capătul de scriere, pot să scrie în canal), și/sau mai multi cititori (*i.e.*, toate procesele ce au acces la capătul de citire, pot să citească din canal).
- Unitatea de informație pentru ambele tipuri de canale este *octetul*.
  Cu alte cuvinte, cantitatea minimă de informație ce poate fi scrisă în canal, respectiv citită din canal, este de 1 octet.
- Capacitatea unui canal de comunicație este limitată la o anumită dimensiune maximă (*e.g.*, 4 Ko, 16 Ko, 64 Ko, ș.a.), ce este configurabilă.
  Spre exemplu, în `Linux` (începând de la versiunea 2.6.35) se poate afla, respectiv configura, capacitatea unui canal de comunicație prin operațiile `F_GETPIPE_SZ`, respectiv `F_SETPIPE_SZ`, disponibile prin apelul de sistem `fcntl`.
  Pentru detalii, consultați documentația acestui apel (*i.e.*, `man 2 fcntl`), precum și exercițiul rezolvat [A pipe's capacity].

Caracteristici și restricții ale canalelor de comunicație (cont.)

- Practic, ambele tipuri de canale (*i.e.*, și cele anonime, și cele cu nume) funcționează ca o coadă, adică o listă FIFO (*First-In, First-Out*), deci citirea din canal se face cu "distrugerea" (*i.e.*, *consumul din canal a*) informației citite (!), iar scrierea în canal se face prin "inserarea" în coadă a informației scrise.

  Concluzie: așadar, citirea dintr-un fișier *fifo* diferă de citirea din fișiere obișnuite, pentru care citirea se face fără consumarea informației din fișier.

- În cazul fișierelor obișnuite am văzut că există noțiunea de *offset* (*i.e.*, poziția curentă în fișier, de la care se efectuează operația curentă de citire sau scriere).

  În schimb, nici pentru fișierele *fifo*, nici pentru canalele anonime nu există această noțiune de *offset*, ele funcționând precum o coadă FIFO.

20 / 41

Comportamentul implicit, de tip blocant

- Citirea dintr-un canal de comunicație funcționează în felul următor:
  - Apelul de citire `read` va citi din canal și va returna imediat, fără să se blocheze, numai dacă mai este suficientă informație în canal, iar în acest caz valoarea returnată reprezintă numărul de octeți citiți din canal.
  - Altfel, dacă canalul este gol, sau nu conține suficientă informație, apelul de citire `read` va rămâne blocat până când va avea suficientă informație în canal pentru a putea citi cantitatea de informație specificată, ceea ce se va întâmpla în momentul când un alt proces va scrie în canal.
  - Alt caz de excepție la citire: dacă un proces încearcă să citească din canal și nici un proces nu mai este capabil să scrie în canal (deoarece toate procesele și-au închis deja capătul de scriere), atunci apelul `read` returnează imediat valoarea `0` prin care se semnalizează că "a citit `EOF`" din canal.
    În concluzie, pentru a se putea citi `EOF` din canal, trebuie ca mai întâi toate procesele să închidă canalul în scriere (adică să închidă descriptorul corespunzător capătului de scriere).

21 / 41

Comportamentul implicit, de tip blocant (cont.)

- Scrierea într-un canal de comunicație funcționează în felul următor:
  - Apelul de scriere `write` va scrie în canal și va returna imediat, fără să se blocheze, numai dacă mai este suficient spațiu liber în canal, iar în acest caz valoarea returnată reprezintă numărul de octeți efectiv scriși în canal (care poate să nu coincidă întotdeauna cu numărul de octeți ce se doreau a se scrie, căci pot apare eventuale erori I/O).
  - Altfel, dacă canalul este plin, sau nu conține suficient spațiu liber, apelul de scriere `write` va rămâne blocat până când va avea suficient spațiu liber în canal pentru a putea scrie informația specificată ca argument, ceea ce se va întâmpla în momentul când un alt proces va citi din canal.
  - Alt caz de excepție la scriere: dacă un proces încearcă să scrie în canal și nici un proces nu mai este capabil să citească din canal (deoarece toate procesele și-au închis deja capătul de citire), atunci sistemul va trimite acelui proces semnalul `SIGPIPE`, ce cauzează terminarea forțată a procesului, fără a afișa însă vreun mesaj de eroare (*Notă*: versiunile mai vechi de Linux afișau "Broken pipe").

22 / 41

Comportamentul implicit, de tip blocant (cont.)

Observație: în locul primitivelor `read` și `write` din API-ul `POSIX`, putem folosi funcțiile I/O de nivel înalt din biblioteca `stdio` pentru a citi din canal (*e.g.*, cu `fread`, `fscanf`, ș.a.) și, respectiv, pentru a scrie în canal (*e.g.*, cu `fwrite`, `fprintf`, ș.a.).

Și aceste funcții de bibliotecă au un *comportament implicit blocant*, similar cu cel descris mai sus, singura diferență fiind aceea că, reamintiți-vă, aceste funcții lucrează *buffer*-izat (!), *i.e.* folosind un *cache* local în *user-space*.

Consecință: modul de lucru *buffer*-izat al funcțiilor I/O din `stdio`, poate cauza uneori erori logice (*i.e.*, *bug*-uri) dificil de depistat, *datorate neatenției programatorului*, care poate uita să forțeze "golirea" *buffer*-ului în canal cu ajutorul funcției `fflush`, imediat după apelul funcției de scriere utilizate pentru a scrie acea informație în canal.

Și astfel, un proces cititor al acelei informații va rămâne blocat în apelul de citire, deoarece informația încă nu a ajuns în canal, iar programatorul va căuta cauza blocajului în altă parte, crezând că informația, pe care o scrisese, a ajuns "instantaneu" (*i.e.*, *fără nicio întârziere sesizabilă*) în canal (!).

Recomandare: acordați mare atenție să nu comiteți acest gen de greșeli logice, căci le-am observat de nenumărate ori, pe parcursul anilor, în programele scrise de studenți.

23 / 41

Comportamentul de tip neblocant

Cele descrise mai devreme, despre blocarea apelurilor de citire, respectiv de scriere, în cazul canalului gol, respectiv plin, corespund comportamentului implicit, de tip ***blocant***, al canalelor de comunicație.

Acest comportament implicit poate fi modificat, pentru ambele tipuri de canale de comunicație, într-un comportament de tip ***neblocant***, situație în care apelurile de citire și, respectiv, de scriere, nu mai rămân blocate în cazul canalului gol și, respectiv, în cazul canalului plin, ci returnează imediat valoarea `-1`, setând în mod corespunzător variabila `errno`.

Mai mult, putem modifica *separat* comportamentul pentru oricare dintre cele două capete ale unui canal, nu suntem limitați doar la a schimba *simultan* comportamentul pentru ambele capete (!).

În plus, în cazul canalelor cu nume, o deschidere *neblocantă* a unuia dintre capetele canalului va reuși imediat, fără să mai aștepte ca vreun alt proces să deschidă celălalt capăt, precum se întâmplă în cazul deschiderii implicite, de tip blocant.

Comportamentul de tip neblocant (cont.)

Modificarea comportamentului implicit în comportament ***neblocant*** se realizează prin setarea atributului `O_NONBLOCK` pentru descriptorul corespunzător acelui capăt al canalului de comunicație pentru care se dorește modificarea comportamentului.

Setarea atributului `O_NONBLOCK` pentru descriptorul dorit, se poate face astfel:

1. fie direct la deschiderea explicită a canalului, *e.g.* printr-un apel de forma:
   fd_out `= open("`*canal_fifo*`", O_WRONLY | O_NONBLOCK );`
   care va seta la deschidere atributul `O_NONBLOCK` doar pentru capătul de scriere.
   Această modalitate este posibilă numai pentru canale cu nume (*i.e.*, fișiere *fifo*).
2. fie după deschiderea, implicită sau explicită, a canalului, utlizând primitiva `fcntl`, *e.g.* printr-un apel de forma: `fcntl(`*fd_out*`, F_SETFL, O_NONBLOCK );`
   Această modalitate este posibilă pentru ambele tipuri de canale.

Temă: scrieți un program prin care să determinați capacitatea ambelor tipuri de canale de comunicație pe sistemul `Linux` pe care lucrați.
Rezolvare: dacă nu reușiți să rezolvați singuri tema, citiți exercițiile rezolvate [A pipe's capacity] și [A fifo's capacity] prezentate în Laboratorul #11.

Agenda

26 / 41

Clasificarea șabloanelor de comunicație inter-procese

După numărul de procese “scriitori” și, respectiv, de procese “cititori” ce utilizează un anumit canal de comunicație (anonim sau cu nume) pentru a comunica între ele, putem diferenția următoarele șabloane de comunicație inter-procese:

- Comunicație *unul-la-unul*: canalul este folosit de un singur proces “scriitor” pentru a transmite date unui singur proces “cititor”.
- Comunicație *unul-la-mulți*: canalul este folosit de un singur proces “scriitor” pentru a transmite date mai multor procese “cititori”.
- Comunicație *mulți-la-unul*: canalul e folosit de mai multe procese “scriitori” pentru a transmite date unui singur proces “cititor”.
- Comunicație *mulți-la-mulți*: canalul e folosit de mai multe procese “scriitori” pentru a transmite date mai multor procese “cititori”.

27 / 41

Șablonul de comunicație *unul-la-unul*

Comunicația *unul-la-unul* reprezintă șablonul cel mai simplu, neridicând probleme deosebite de implementare. Din acest motiv, este și cel mai folosit în practică.

Notă: exemplele de programe date anterior, în secțiunea despre canale anonime, se încadrează în acest șablon de comunicație.

Demo: exercițiul rezolvat ['Producer-consumer' pattern #1, using fifos for IPC] din Laboratorul #11 prezintă două programe care, fiecare în parte, ilustrează folosirea unui canal cu nume pentru comunicația unul-la-unul între două procese, unul cu rol de producător, iar celălalt cu rol de consumator.

Celelalte trei șabloane ridică anumite probleme de sincronizare, datorate accesului concurent al mai multor procese la câte unul, sau la ambele, dintre capetele canalului, probleme de care trebuie să se țină cont la implementarea lor.

Vom trece în revistă, pe rând, aceste probleme de sincronizare, ce pot avea efecte asupra integrității datelor (*e.g.*, "coruperea" mesajelor) . . .

28 / 41

Șablonul de comunicație *unul-la-mulți*

Factori ce pot genera anumite probleme de sincronizare, cu efecte asupra integrității datelor:

- *lungimea* mesajelor:
  - mesaje de lungime *constantă*
    Nu ridică probleme deosebite – fiecare mesaj poate fi citit *atomic* (*i.e.*, dintr-o dată, printr-un singur apel `read`).
  - mesaje de lungime *variabilă*
    Pot apare probleme de sincronizare, deoarece mesajele nu mai pot fi citite *atomic*. Soluția este folosirea mesajelor formatate astfel:

    MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

    header-ul fiind un mesaj de lungime fixă ce conține lungimea mesajului propriu-zis.
    Protocol de comunicație utilizat: sunt necesare 2 apeluri `read` pentru a citi un mesaj în întregime, de aceea trebuie garantat accesul exclusiv la canal (folosind, de exemplu, blocaje pe fișiere).

29 / 41

Șablonul de comunicație *unul-la-mulți* (cont.)

Factori ce pot genera anumite probleme de sincronizare, cu efecte asupra integrității datelor:

- *destinatarul* mesajelor:
  - mesaje cu destinatar *oarecare*
    Nu ridică probleme deosebite – fiecare mesaj poate fi citit și prelucrat de oricare dintre procesele “cititori”.
  - mesaje cu destinatar *specificat*
    Trebuie asigurat faptul că mesajul este citit exact de către “cititorul” căruia îi era destinat.
    Soluția – am putea folosi mesaje formatate astfel:

    MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

    header-ul conținând un identificator al destinatarului.
    Pentru citire, se poate aplica protocolul de comunicație discutat la mesaje de lungime variabilă. Însă, apare o *problemă suplimentară*: dacă un “cititor” a citit un mesaj care nu-i era destinat lui, cum facem să-l livrăm celui căruia îi era destinat? O soluție ar fi să îl scrie înapoi în canal, și apoi va face o pauză aleatoare înainte de a încerca să citească din nou din canal. *Notă*: această soluție poate suferi de fenomenul *starvation*.

30 / 41

Șablonul de comunicație *mulți-la-unul*

Factori ce pot genera anumite probleme de sincronizare, cu efecte asupra integrității datelor:

- *lungimea* mesajelor:
  - mesaje de lungime *constantă*
    Nu ridică probleme deosebite – fiecare mesaj poate fi scris *atomic* (*i.e.*, dintr-o dată, printr-un singur apel `write`).
  - mesaje de lungime *variabilă*
    Trebuie indicată “cititorului” lungimea fiecărui mesaj. Soluția este folosirea mesajelor formatate astfel:

    MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

    header-ul fiind un mesaj de lungime fixă ce conține lungimea mesajului propriu-zis.
    Nu ridică probleme deosebite – fiecare mesaj, astfel formatat, poate fi scris *atomic*, printr-un singur apel `write`, deci nu trebuie garantat accesul exclusiv la canal.

31 / 41

Șablonul de comunicație *mulți-la-unul* (cont.)

Factori ce pot genera anumite probleme de sincronizare, cu efecte asupra integrității datelor:

- *destinatarul* mesajelor:
  - mesaje cu expeditor *oarecare*
    Nu ridică probleme deosebite – fiecare mesaj poate fi citit de procesul "cititor" și prelucrat în același fel, indiferent de la care dintre procesele "scriitori" provine.
  - mesaje cu expeditor *specificat*
    Trebuie asigurat că mesajul îi indică "cititorului" care este "scriitorul" care i l-a trimis. Soluția – mesaje formatate în felul următor:

    MESAJ = HEADER + MESAJUL PROPRIU-ZIS ,

    header-ul conținând un identificator al expeditorului.
    Notă: scrierea mesajului astfel formatat se va face printr-un singur apel `write`, la fel ca la mesaje de lungime variabilă.

32 / 41

Șablonul de comunicație *mulți-la-mulți*

Problemele de sincronizare ce pot apare în cazul acestui șablon, pot fi cauzate de oricare dintre factorii discutați la șabloanele *unul-la-mulți* și *mulți-la-unul*:

- *lungimea* mesajelor
- *expeditorul* mesajelor
- *destinatarul* mesajelor

Tratarea acestora se poate face prin combinarea soluțiilor prezentate la șabloanele de comunicație *unul-la-mulți* și *mulți-la-unul*.

Observație: pentru simplitate, se poate prefera uneori înlocuirea unui singur canal folosit pentru comunicație *unul-la-mulți*, cu mai multe canale folosite pentru comunicație *unul-la-unul*, *i.e.* cu câte un canal pentru fiecare proces "cititor" existent.
Similar se poate proceda și pentru șabloanele *unul-la-mulți* și *mulți-la-mulți*.

Demo: a se vedea programele `suma_pipes.c` și `suma_fifos.c` ([2]), care reprezintă rescrieri ale programului `suma_files.c` din lecția despre fork(), prin înlocuirea fișierelor obișnuite cu canale (anonime și, respectiv, cu nume) pentru comunicațiile dintre supervisor și workeri. Comunicațiile dinspre workeri spre supervisor folosesc șablonul "mulți-la-unul". În schimb, șablonul "unul-la-mulți" pentru comunicațiile dinspre supervisor spre workeri l-am implementat pe baza observației de mai sus.

33 / 41

Agenda

34 / 41

Aplicația #1: implementarea unui semafor

Cum am putea implementa un semafor folosind canale *fifo* ?

O posibilă implementare ar consta în următoarele idei:

Inițializarea semaforului s-ar realiza prin crearea unui fișier *fifo* de către un proces cu rol de *supervizor* (acesta poate fi oricare dintre procesele ce vor folosi acel semafor, sau poate fi un proces separat).

Acest proces *supervizor* va scrie inițial în canal 1 octet oarecare, dacă e vorba de un semafor binar (sau n octeți oarecare, dacă e vorba de un semafor general n-ar).
Iar apoi va păstra deschise ambele capete ale canalului pe toată durata de execuție a proceselor ce vor folosi acel semafor (cu scopul de a nu se pierde pe parcurs informația din canal, datorită inexistenței la un moment dat a măcar unui proces care să aibă deschis măcar vreunul dintre capete, conform celor discutate anterior legat de perioada de retenție a informației într-un canal *fifo*).

35 / 41

Aplicația #1: implementarea unui semafor (cont.)

Operația `wait` va consta în citirea unui octet din fișierul *fifo*.
Mai precis, întâi se va face deschiderea lui, urmată de citirea efectivă a unui octet, și apoi eventual închiderea fișierului.

Operația `signal` va consta în scrierea unui octet în fișierul *fifo*.
Mai precis, întâi se va face deschiderea lui, urmată de scrierea efectivă a unui octet, și apoi eventual închiderea fișierului.

Observații:
i) citirea se va face, în modul implicit, *blocant*, ceea ce va asigura așteptarea procesului la punctul de intrare în secțiunea sa critică atunci cînd semaforul este "pe roșu", adică dacă canalul *fifo* este gol.
ii) scrierea nu se va putea bloca (cu condiția ca n-ul semaforului general să nu depășească capacitatea maximă pe care o putem configura pentru un canal).

Temă: implementați în C un semafor binar pe baza ideilor de mai sus și scrieți un program demonstrativ în care să utilizați semaforul astfel implementat pentru asigurarea excluderii mutuale a unei secțiuni critice de cod (pentru "inspirație" în scrierea programului demonstrativ, revedeți problemele de sincronizare discutate în cursurile teoretice).

36 / 41

Aplicația #2: implementarea unei aplicații de tip client/server

O *aplicație de tip client/server* este compusă din două componente:

- **serverul**: este un program care dispune de un anumit număr de *servicii* (*i.e.* funcții/operații), pe care le pune la dispoziția clienților.
- **clientul**: este un program care "interoghează" serverul, solicitându-i *efectuarea unui serviciu* (dintre cele puse la dispoziție de acel server).

Exemplu: *Browser*ele pe care le folosiți pentru a naviga pe INTERNET sunt un exemplu de program client, care se conectează la un program server, numit *server de web*, solicitându-i transmiterea unei pagini *web*, care apoi este afișată în fereastra grafică a *browser*ului.

Implementarea unei aplicații de tip client-server o puteți face în felul următor:

Programul server va fi rulat în *background*, și va sta în așteptarea cererilor din partea clienților, putând servi mai mulți clienți simultan.
Iar clienții vor putea fi rulați mai mulți simultan (din același cont și/sau din conturi utilizator diferite), și se vor "conecta" la serverul rulat în *background*.

Notă: pot exista, la un moment dat, mai multe procese client care încearcă, fiecare independent de celelalte, să folosească serviciile puse la dispoziție de procesul server.

37 / 41

Aplicația #2: implementarea unei aplicații de tip client/server (cont.)

Observație: în realitate, programul server este rulat pe un anumit calculator, iar clienții pe diverse alte calculatoare, conectate la INTERNET, comunicația realizându-se folosind *socket*-uri, prin intermediul rețelelor de calculatoare.
Însă puteți simula această "realitate" folosind comunicație prin canale cu nume și executând toate procesele (*i.e.*, serverul și clienții) pe un același calculator, eventual din conturi utilizator diferite.

Tipurile de servere existente în realitate, d.p.d.v. al servirii "simultane" a mai multor clienți, se împart în două categorii:

- server *iterativ*
  Cât timp durează efectuarea unui serviciu (*i.e.*, rezolvarea unui client), serverul este blocat: nu poate răspunde cererilor venite din partea altor clienți. Deci nu poate rezolva mai mulți clienți în același timp!
- server *concurent*
  Pe toată durata de timp necesară pentru efectuarea unui serviciu (*i.e.*, rezolvarea unui client), serverul nu este blocat, ci poate răspunde cererilor venite din partea altor clienți. Deci poate rezolva mai mulți clienți în același timp!

38 / 41

Aplicația #2: implementarea unei aplicații de tip client/server (cont.)

Detalii legate de implementare:

- Pentru implementarea unui server de tip iterativ este suficient un singur proces secvențial. În schimb, pentru implementarea unui server de tip concurent este nevoie de mai multe procese secvențiale: un proces *supervisor*, care așteaptă sosirea cererilor din partea clienților, și la fiecare cerere sosită, el va crea un nou proces fiu, un *worker* care va fi responsabil cu rezolvarea propriu-zisă a clientului respectiv, iar *supervisor*-ul va relua imediat așteptarea sosirii unei noi cereri, fără să aștepte terminarea procesului fiu. (Sau, alternativ, se poate implementa printr-un singur proces *multi-threaded*.)
- Pentru comunicarea între procesele client și procesul server este necesar să se utilizeze, drept canale de comunicație, fișiere *fifo*. (*Motivul*: nu se pot folosi canale anonime deoarece procesul server și procesele clienți nu sunt înrudite prin `fork`/`exec`.)
- Permisiunile fișierelor *fifo* folosite pentru comunicație trebuie configurate adecvat, astfel încât să permită execuția proceselor client din *conturi utilizator diferite*.

39 / 41

Aplicația #2: implementarea unei aplicații de tip client/server (cont.)

- Un alt aspect legat tot de comunicație: serverul nu cunoaște în avans clienții ce se vor conecta la el pentru a le oferi servicii, în schimb clientul trebuie să cunoască serverul la care se va conecta pentru a beneficia de serviciul oferit de el.

  Ce înseamnă aceasta d.p.d.v. practic?

  Serverul va crea un canal *fifo* cu un nume fixat, cunoscut în programul client, și va aștepta sosirea informațiilor pe acest canal.
  Un client oarecare se va conecta la acest canal *fifo* cunoscut și va transmite informații de identificare a sa, care vor fi folosite ulterior pentru realizarea efectivă a comunicațiilor implicate de serviciul solicitat (s-ar putea să fie nevoie de canale suplimentare, particulare pentru acel client, pentru a nu se "amesteca" între ele comunicațiile destinate unui client cu cele destinate altui client conectat la server în același timp cu primul).

Temă: implementați un joc *multi-player* "în rețea", pe baza ideilor descrise mai sus.

40 / 41

Referințe bibliografice

41 / 41

Bibliografie obligatorie

[1] Capitolul 5, §5.1, §5.2, §5.3 și §5.5 din cartea "Sisteme de operare – manual pentru ID", autor C. Vidrașcu, editura UAIC, 2006. Acest manual este accesibil, în format PDF, din pagina disciplinei "Sisteme de operare":

- https://profs.info.uaic.ro/~vidrascu/SO/books/ManualID-SO.pdf

[2] Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la adresele:

- https://profs.info.uaic.ro/~vidrascu/SO/cursuri/C-programs/pipe/
- https://profs.info.uaic.ro/~vidrascu/SO/cursuri/C-programs/fifo/

[3] POSIX API: `man 2 pipe`, `man 2 mkfifo`.

41 / 41