Programare de sistem în C pentru platforma Linux (III)

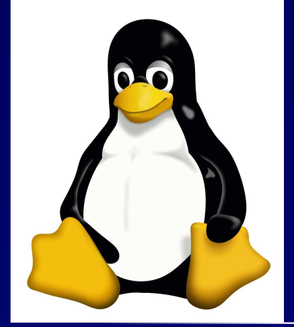
Gestiunea fișierelor, partea a III-a:

Fișiere mapate în memorie – primitiva `mmap()`

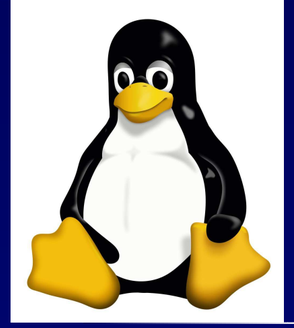
Cristian Vidrașcu

cristian.vidrascu@info.uaic.ro

Aprilie, 2024



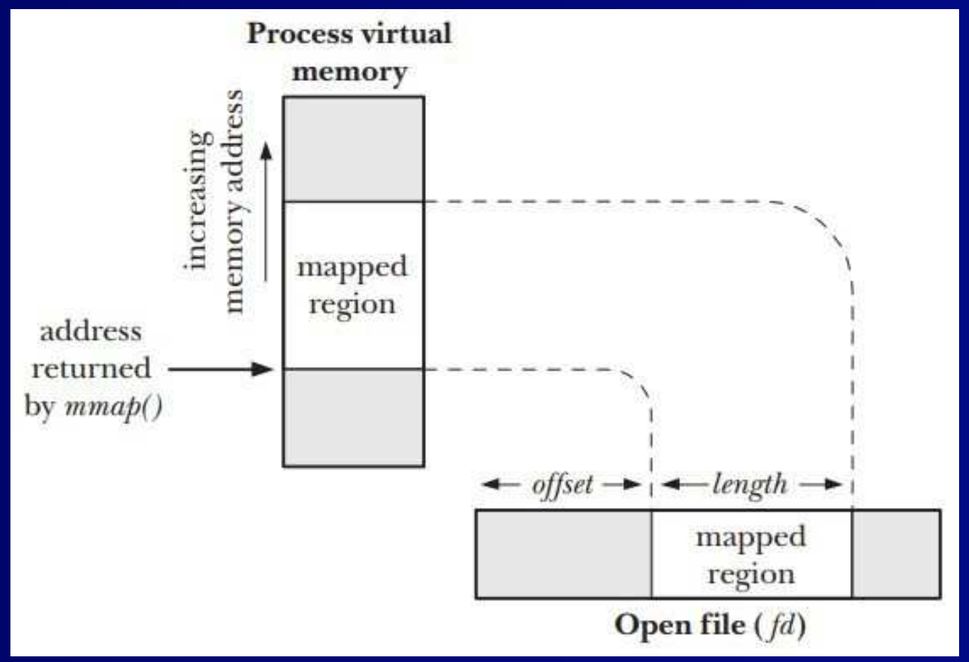
Sumar



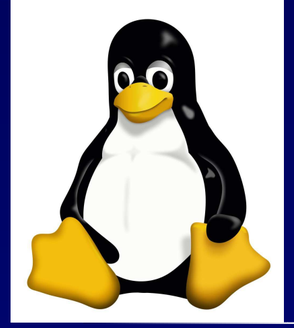
Introducere

Fișier mapat în memorie – un mecanism prin care (o parte din) conținutul unui fișier este “mapat” în memorie, în spațiul virtual de adrese al procesului ce apelează, în acest scop, primitiva `mmap`.

Prin această “mapare” se realizează practic o corelație directă “octet-la-octet” între o porțiune din spațiul virtual de adrese al procesului și o porțiune a unui fișier de pe disc:



Cu alte cuvinte, paginilor virtuale ce formează respectiva porțiune din spațiul virtual de adrese al procesului, li se asociază drept *backing store* (*i.e.*, “spațiul” pe disc rezervat pentru evacuarea lor din memorie), de către nucleul sistemului de operare, zona de pe disc ce stochează acea porțiune a fișierului de pe disc, în loc de a le rezerva spațiu în *fișierul de swap* al sistemului de operare. *Observație*: veți afla mai multe detalii despre administrarea memoriei virtuale prin *paginare la cerere* în cursul teoretic #10 .



Introducere (cont.)

Atenție: termenul *fișier mapat în memorie* (în engleză, *memory-mapped file*) se referă la acea porțiune din spațiul virtual de adrese al procesului pentru care s-a stabilit, printr-un apel `mmap`, o corelație directă “octet-la-octet” cu o porțiune a unui fișier de pe disc. Deci nu confundați semnificația acestui termen cu fișierul propriu-zis de pe disc (sau cu porțiunea acestuia de pe disc).

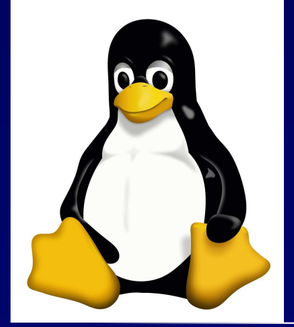
* * *

Printr-o mapare, putem face accese de citire și scriere direct în memorie asupra fișierului, ca și cum am citi sau scrie diverse variabile din program, fără să mai utilizăm apelurile de sistem `read`/`write` (sau funcțiile de I/O din biblioteca standard de C).
Efectul scrierilor în memorie va fi “propagat” pe disc cu întârziere, atunci când nucleul decide să salveze paginile *dirty* pe disc (*e.g.*, atunci când le selectează drept victime pentru evacuare din memorie).

* * *

Un alt avantaj al acestui mecanism: un anumit fișier poate fi “mapat” simultan în spațiile virtuale de adrese a două (sau mai multor) procese și astfel procesele respective pot coopera schimbând informații prin modelul de comunicație cu *shared memory*.

Un exemplu simplu de procese cooperante prin modelul de comunicație cu memorie partajată: revedeți șablonul producător-consumator, discutat în cursul teoretic #6.
Alte exemple de procese cooperante prin modelul de comunicație cu memorie partajată: revedeți toate problemele de sincronizare discutate în cursurile teoretice #5 și #6.



Agenda

Introducere

Primitivele din familia mmap

- Primitiva mmap
- Primitiva munmap
- Caracteristici ale mapărilor create cu mmap
- Primitiva msync
- Mapări ne-persistente

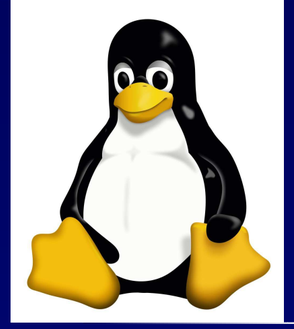
Demo: programe cu mmap

- Exemplul #1: O mapare “privată”, cu permisiuni *read-only*
- Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*
- Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării
- Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier
- Exemplul #5: O mapare ne-persistentă cu nume
- Exemplul #6: O mapare ne-persistentă anonimă
- Alte exemple de programe cu mapări

Procese cooperante

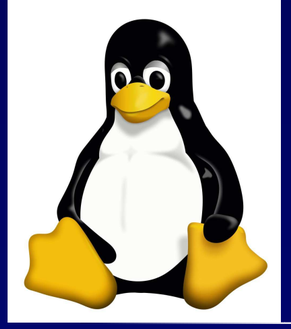
- Modele de comunicație între procese (IPC)
- Șabloane arhitecturale de cooperare și sincronizare
- Semafoare POSIX

Referințe bibliografice



Primitiva `mmap`

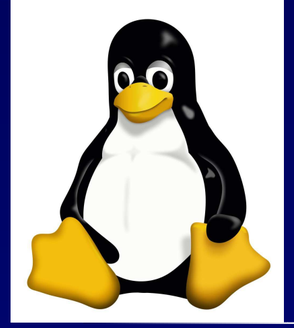
- *“Maparea” unui fișier în memoria virtuală a unui proces* : se realizează cu primitiva `mmap`.
  Interfața funcției `mmap` ([4]) :
  `void *mmap(void*addr, size_t length, int prot, int flags, int fd, off_t offset)`
  unde:
  - **Valoarea returnată**: adresa de start a mapării create cu succes (*i.e.*, *începutul regiunii mapate în spațiul virtual al procesului apelant*), sau `MAP_FAILED` (= `(void*)-1`) în caz de eroare.
  - *addr* = adresa de start pentru noua mapare ce se va crea în spațiul virtual al procesului apelant. Dacă *addr*=`NULL`, nucleul va alege în mod convenabil o adresă *page-aligned* (*i.e.*, multiplu de dimensiunea paginii) la care va crea noua mapare. Altfel, valoarea *addr* este folosită de nucleu doar cu rol de *hint* (cu o excepție: în cazul folosirii *flag*-ului `MAP_FIXED`).
  - *length* = lungimea noii mapări ce se creează (lungimea trebuie să fie un întreg strict pozitiv).
  - *fd* = identifică fișierul (sau un alt obiect, *e.g.* un *device*) asociat mapării ce se creează.
    Notă: descriptorul *fd* poate fi închis **imediat** după apelul `mmap`, fără invalidarea mapării create.
  - *offset* = trebuie să fie un întreg pozitiv multiplu de dimensiunea paginii (!).
    Notă: maparea nou creată este **inițializată** prin copierea de pe disc a conținutului porțiunii din fișierul asociat ce începe de la poziția *offset* și de lungime *length* (cu o excepție: în cazul folosirii *flag*-ului `MAP_UNINITIALIZED`). Iar ca “destinație pe disc” pentru acele modificări efectuate în memorie ce **trebuie “propagate”** pe disc este folosită aceeași porțiune din fișier.



Primitiva `mmap` (cont.)

- *“Maparea” unui fișier în memoria virtuală a unui proces* – interfața funcției `mmap` (cont.) :
  - *`prot`* = specifică tipul de protecție al tuturor paginilor de memorie ce formează noua mapare (și trebuie să nu fie în conflict cu modul de deschidere al fișierului). Poate avea ca valoare fie constanta simbolică `PROT_NONE` – paginile mapării nou create NU vor putea fi accesate, fie o combinație (*i.e.*, disjuncție logică pe biți) a uneia sau a mai multora dintre constantele:
    - `PROT_READ` – paginile mapării nou create vor putea fi accesate pentru citire ;
    - `PROT_WRITE` – paginile mapării nou create vor putea fi accesate pentru scriere ;
    - `PROT_EXEC` – paginile mapării nou create vor putea fi accesate pentru execuție .
  - *`flags`* = o serie de *flag*-uri folosite pentru a determina dacă modificările (scrierile) efectuate de proces în paginile mapării vor fi “vizibile” sau nu și în celelalte procese ce mapează același fișier, precum și dacă aceste modificări efectuate în memorie vor fi “propagate” (*i.e.*, *flush*-uite pe disc) în fișierul propriu-zis stocat pe disc. Poate fi folosită exact una singură dintre valorile:
    - `MAP_PRIVATE` – se creează o mapare “privată” (de tipul *copy-on-write*) ;
    - `MAP_SHARED` – se creează o mapare “partajată” .

    Aceasta poate fi însoțită, eventual, de o combinație (*i.e.*, disjuncție logică pe biți) a altor valori, precum ar fi: `MAP_FIXED`, `MAP_LOCKED`, `MAP_ANONYMOUS`, `MAP_UNINITIALIZED`, ș.a. Pentru a afla semnificația acestor valori, consultați documentația funcției `mmap` ([4]).



Primitiva `munmap`

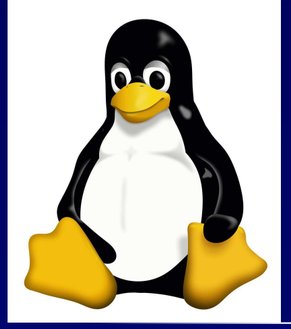
- *“Ștergerea” unei mapări din memoria virtuală a unui proces* : se realizează cu primitiva `munmap`.
  Interfața funcției `munmap` ([4]) :

  ```
  int munmap(void *addr, size_t length)
  ```

  - *addr* = adresa de start pentru maparea din spațiul virtual al procesului apelant ce se va șterge. Adresa specificată trebuie să fie multiplu de dimensiunea paginii.
  - *length* = lungimea mapării ce se va șterge.
  - **Valoarea returnată**: `0`, în caz de succes, și respectiv `-1`, în caz de eroare.

Observații:

1) Parametrul *length* nu trebuie să fie neapărat multiplu de dimensiunea paginii, dar se va lua în considerare cel mai mic multiplu de dimensiunea paginii, mai mare sau egal cu *length* (deoarece unitatea de alocare / dealocare în spațiul virtual de adrese al unui proces este pagina).
2) Apelul `munmap` “șterge” intervalul de adrese specificat prin parametri (rotunjit la un număr întreg de pagini) din spațiul virtual de adrese al procesului apelant. Aceasta are ca efect faptul că orice acces ulterior la vreo adresă din acel interval va genera o eroare de tip “referință invalidă” (*i.e.*, se generează semnalul `SIGSEGV`, ceea ce cauzează terminarea anormală a procesului, cu un mesaj de eroare “Segmentation fault”).
3) Nu este eroare dacă maparea ce se șterge nu reprezintă un interval de adrese corespunzătoare unor pagini mapate în spațiul virtual de adrese al procesului apelant, la momentul apelului `munmap` respectiv.
4) Mapările create prin apeluri `mmap` sunt “șterse” automat la terminarea execuției procesului. Pe de altă parte, închiderea descriptorului de fișier utilizat într-un apel `mmap` anterior nu provoacă “ștergerea” mapării respective.



Caracteristici ale mapărilor create cu `mmap`

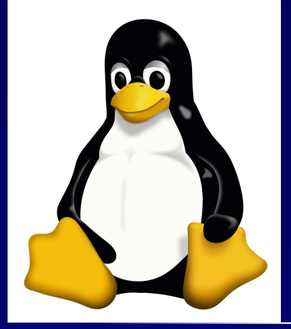
1) *Important*: modul portabil de a crea o mapare este de a specifica *`addr`* ca `0` (`NULL`) și de a omite `MAP_FIXED` din *`flags`*. În acest caz, nucleul alege adresa pentru mapare; adresa va fi aleasă într-o manieră adecvată pentru a nu intra în conflict cu nicio mapare existentă și nu va fi 0.

2) Semnificația celor două tipuri de mapări (`MAP_PRIVATE` vs. `MAP_SHARED`) :

   — Pentru o mapare “privată” (de tipul *copy-on-write*), scrierile efectuate de procesul ce a creat-o NU vor fi “vizibile” în celelalte procese ce mapează aceeași porțiune de fișier și nici NU vor fi “propagate” în fișierul propriu-zis de pe disc (ci doar, eventual, în *fișierul de swap* al sistemului).

   — Pentru o mapare “partajată”, scrierile efectuate de proces vor fi “vizibile” în celelalte procese ce mapează aceeași porțiune de fișier și vor fi “propagate” în fișierul propriu-zis de pe disc.
   Important: momentul “propagării” pe disc a scrierilor în memorie este, implicit, controlat de către nucleu, prin algoritmul de gestiune a paginilor *dirty*. Însă, putem forța explicit “propagarea” pe disc a scrierilor în memorie folosind primitiva `msync`.

3) Lungimea efectivă (*i.e.*, dimensiunea în octeți) a mapării nou create va fi cel mai mic multiplu de dimensiunea paginii, mai mare sau egal cu *`length`* (deoarece unitatea de alocare / dealocare în spațiul virtual de adrese al unui proces este pagina).
   Astfel, dacă parametrul *`length`* nu este multiplu de dimensiunea paginii, atunci la crearea mapării restul adreselor din ultima pagină a mapării vor fi inițializate cu zero, iar scrierile ulterioare la aceste adrese nu vor da eroare, dar nici NU vor fi “propagate” în fișierul de pe disc.



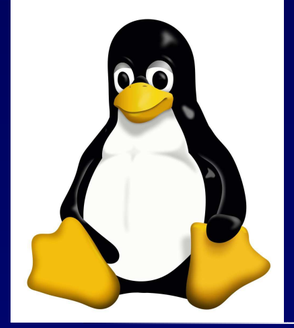
Caracteristici ale mapărilor create cu `mmap` (cont.)

4) În urma unui apel `fork`, procesul fiu “moștenește” memoria mapată cu primitiva `mmap` de către părinte, anterior creării fiului. Maparea respectivă va avea în procesul fiu aceleași atribute și aceeași porțiune de fișier asociată ca în procesul părinte (mai multe detalii despre aceste aspecte vom vedea în lecția practică următoare, dedicată apelului `fork`).

5) Pe marea majoritate a arhitecturilor hardware (*e.g.*, arhitectura x86 / x64) modelul de protecție a acceselor la memorie permite doar valorile *read-only* și *read & write*, dar nu și *write-only*.
Cu alte cuvinte, permisiunea `PROT_WRITE` implică automat și permisiunea `PROT_READ`, chiar dacă aceasta din urmă nu este specificată explicit în apelul `mmap`.

6) Pe anumite arhitecturi hardware permisiunea `PROT_READ` implică automat și permisiunea `PROT_EXEC` (*e.g.*, *CPU*-uri x86 mai vechi, fără suport pentru bitul NX, ș.a.), iar pe alte arhitecturi nu implică acest lucru (*e.g.*, arhitectura x64, *CPU*-uri x86 cu suport pentru bitul NX, ș.a.).
Pentru portabilitatea programelor, este recomandat să se specifice explicit permisiunea `PROT_EXEC` în apelul `mmap` ce va crea o mapare din care se intenționează să se execute cod.

7) Pentru a modifica protecția paginilor dintr-o mapare, se poate utiliza primitiva `mprotect` ([4]).

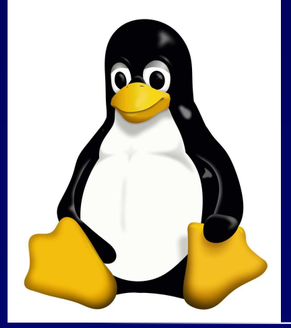


Caracteristici ale mapărilor create cu `mmap` (cont.)

8) Paginile fizice (din RAM) ce stochează paginile virtuale din care este format spațiul virtual de adrese al unui proces sunt gestionate de nucleu conform mecanismului de **administrare a memoriei virtuale prin paginare la cerere** (a se vedea cursurile teoretice #9 și #10 ; până atunci, vă recomand să citiți secțiunile `NOTES` din paginile `man 2 mremap` și `man 2 mmap`).

   Mai exact, pe întreaga durată de viață a unui proces, fiecare pagină virtuală a sa trece prin perioade când este rezidentă în memorie (*i.e.*, se află într-o pagină fizică din RAM) și perioade când nu este rezidentă în memorie (*i.e.*, conținutul său este doar pe disc, într-un fișier mapat în memorie sau în *fișierul de swap* al sistemului).
   Pentru a afla care pagini virtuale sunt rezidente și care nu sunt rezidente la un moment dat, se poate utiliza primitiva `mincore` ([4]).

9) De asemenea, nucleul permite “încuierea” unor pagini virtuale în memorie – astfel, ele vor rămâne rezidente în permanență (până la terminarea procesului sau până la “descuierea” lor), nemaifiind alese drept victimă de algoritmul de *page-swapping*.
   Pentru a “încuia” anumite pagini virtuale ale procesului, sau pe toate, se pot utiliza primitivele `mlock` și, respectiv, `mlockall` ([4]).
   Iar pentru a le “descuia” se pot utiliza primitivele `munlock` și, respectiv, `munlockall` ([4]).



Primitiva `msync`

- *“Sincronizarea” unui fișier cu maparea sa din memoria virtuală a unui proces* : se realizează cu primitiva `msync`. Interfața funcției `msync` ([4]) :

  ```
  int msync(void *addr, size_t length, int flags)
  ```

  — *addr* = adresa de start pentru maparea (din spațiul virtual al procesului apelant) pentru care vrem să “propagăm” pe disc (în porțiunea de fișier asociată mapării) scrierile deja efectuate în memorie și încă “nepropagate” (*i.e.*, paginile *dirty* ale mapării respective).

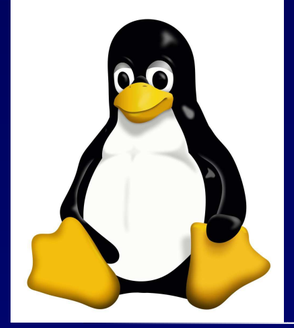
  — *length* = lungimea mapării, și a porțiunii de fișier de pe disc asociate ei, ce se vor sincroniza.

  — *flags* = apelul inițiază, în mod blocant sau neblocant, un *flushing* pe disc a paginilor *dirty* din acea mapare, prin specificarea exact a uneia dintre valorile:

  - `MS_SYNC` – se cere un *flushing* în mod blocant (*i.e.*, se așteaptă finalizarea scrierii efective pe disc a paginilor *dirty* din acea mapare) ;
  - `MS_ASYNC` – se cere un *flushing* în mod neblocant (*i.e.*, fără a se aștepta finalizarea scrierii efective pe disc a paginilor *dirty* din acea mapare) .

  Oricare dintre cele două valori poate fi, eventual, combinată (*i.e.*, disjuncție logică pe biți) cu valoarea `MS_INVALIDATE`, prin care se cere invalidarea celorlalte mapări posibil existente ale aceluiași fișier (prin invalidare, acestea se vor actualiza cu modificările survenite pe disc).

  — **Valoarea returnată**: `0`, în caz de succes, și respectiv `-1`, în caz de eroare.

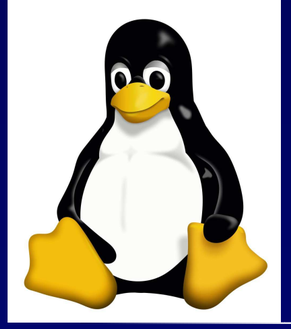


Primitiva `msync` (cont.)

- *“Sincronizarea” unui fișier cu maparea sa din memoria virtuală a unui proces* (cont.)

  Observații:

  1) Parametrul *addr* este valoarea returnată de apelul `mmap` ce a creat acea mapare (deci obligatoriu este multiplu de dimensiunea paginii).
  2) Parametrul *length* este valoarea declarată în apelul `mmap` respectiv, nefiind obligatoriu să fie multiplu de dimensiunea paginii (a se vedea cele explicate anterior).
  3) Reformulez o afirmație anterioară – caracteristica 3) descrisă la apelul `mmap` : dacă parametrul *length* nu este multiplu de dimensiunea paginii, atunci scrierile în maparea din memorie a acelei porțiuni de fișier, la adrese situate în ultima pagină alocată mapării, “dincolo” de adresa dată de restul împărțirii întregi a valorii *length* la dimensiunea paginii, vor reuși fără a da eroare, dar efectele acestor scrieri nu vor fi “propagate” și în fișierul de pe disc.
  4) *Important*: apelul `munmap` nu efectuează și un apel `msync` implicit (*i.e.*, nu face și *flushing* pentru paginile *dirty* din acel moment).
     Cu alte cuvinte, când ștergeți explicit o mapare fără să o sincronizați mai întâi pe disc, este posibil să “pierdeți” ultimele modificări efectuate în memoria acelei mapări (*i.e.*, acestea nu se vor salva în fișierul de pe disc).



Mapări ne-persistente

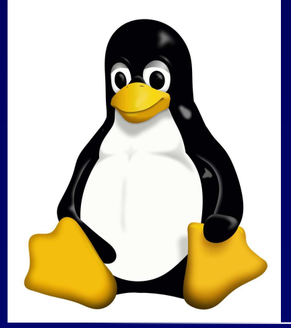
Există două tipuri de fișiere mapate în memorie :

1) Mapări persistente de fișiere :
   Acesta este tipul de mapare implicit, numit și *file-backed mapping*. O mapare de fișier persistentă este asociată cu un fișier sursă de pe un disc și mapează o zonă a memoriei virtuale a procesului către acel fișier de pe disc (*i.e.*, citirea acelor zone ale memoriei cauzează fișierul asociat să fie citit și scrierea acelor zone de memorie provoacă, în cele din urmă, o scriere a fișierului de pe disc).

2) Mapări ne-persistente :
   O mapare ne-persistentă nu este asociată cu vreun fișier de pe disc, *i.e.* mapează o zonă din memoria virtuală a procesului care nu are asociată niciun fișier de pe disc. Conținutul este inițializat la zero. Când ultimul proces a terminat de lucrat cu o astfel de mapare, datele se pierd. Acest tip de mapări este utilizat pentru crearea memoriei partajate pentru comunicații între procese (IPC). Mapările ne-persistente sunt de două feluri :

   — Mapări ne-persistente anonime : acestea sunt create folosind flagul `MAP_ANONYMOUS` într-un apel `mmap` și pot fi utilizate pentru IPC doar între procese înrudite prin apelul `fork`, despre care vom discuta în lecția practică următoare (vom vedea tot atunci și exemple de IPC folosind mapări anonime). *Demo*: a se vedea Exemplul #6 din secțiunea următoare.

   — Mapări ne-persistente cu nume : a se vedea următorul *slide*.



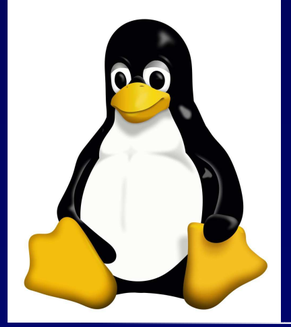
Mapări ne-persistente (cont.)

Mapări ne-persistente cu nume :

API-ul POSIX pentru memorie partajată ([5]) permite proceselor să comunice informații prin partajarea unei regiuni de memorie. Este format din următoarele funcții:

- `shm_open` : Creează și deschide un obiect nou de memorie partajată, sau deschide un obiect existent. Returnează un descriptor de fișier utilizat de către celelalte funcții enumerate mai jos.
- `ftruncate` : Setează dimensiunea obiectului. (Un obiect nou creat are dimensiunea zero.)
- `mmap` / `munmap` : Mapează / de-mapează obiectul de memorie partajată în / din spațiul de adrese virtuale al procesului apelant.
- `fstat` : Returnează o structură `stat` care descrie obiectul de memorie partajată.
- `fchmod` / `fchown` : Pentru a modifica permisiunile / proprietarul unui obiect de memorie partajată.
- `close` : Închide descriptorul de fișier alocat de `shm_open` când nu mai este necesar.
- `shm_unlink` : Elimină din sistem un obiect de memorie partajată (specificat prin numele său).

Persistență : obiectele de memorie partajată POSIX au *persistență la nivel de kernel* – un obiect de memorie partajată va exista până când sistemul este oprit / repornit, sau până când toate procesele au demapat obiectul și a fost șters cu `shm_unlink`.
Accesarea obiectelor de memorie partajată prin sistemul de fișiere : în `Linux`, obiectele de memorie partajată sunt create într-un sistem de fișiere virtual de tip `tmpfs`, montat de obicei sub `/dev/shm`.
Demo: a se vedea Exemplul #5 din secțiunea următoare.



Agenda

Introducere

Primitivele din familia mmap

- Primitiva mmap
- Primitiva munmap
- Caracteristici ale mapărilor create cu mmap
- Primitiva msync
- Mapări ne-persistente

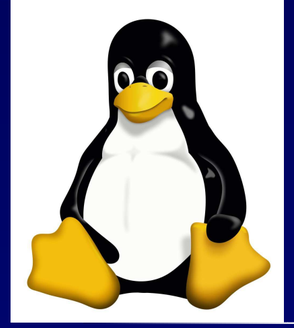
Demo: programe cu mmap

- Exemplul #1: O mapare “privată”, cu permisiuni *read-only*
- Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*
- Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării
- Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier
- Exemplul #5: O mapare ne-persistentă cu nume
- Exemplul #6: O mapare ne-persistentă anonimă
- Alte exemple de programe cu mapări

Procese cooperante

- Modele de comunicație între procese (IPC)
- Șabloane arhitecturale de cooperare și sincronizare
- Semafoare POSIX

Referințe bibliografice



Exemplul #1: O mapare “privată”, cu permisiuni *read-only*

Un exemplu ce ilustrează folosirea apelului `mmap` pentru realizarea unei mapări de tip “privată”, cu permisiuni de acces *read-only*, a unei porțiuni specificate dintr-un fișier.

A se vedea variantele de program `mmap_ex1a.c` și `mmap_ex1b.c` ([1]).

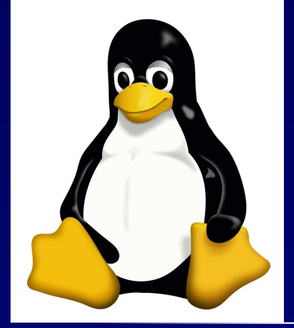
Observație: pentru explicații mai detaliate ale acestor programe și descrierea comportamentului lor la execuție, consultați exemplul [FirstDemo – mmap_ex1] prezentat în suportul de laborator #9.

Ambele variante de program demonstrează *citirea direct din memorie* a informației mapate din fișier, în locul utilizării interfeței clasice de acces I/O la disc (adică fără a folosi apelurile de sistem `read` și `write`, sau funcții de I/O din biblioteci precum `libc`).

Diferența dintre cele două variante de program constă în modul de tratare a cazului în care utilizatorul programului introduce date de intrare “invalide” (*i.e.*, pentru acest program, aceasta înseamnă introducerea unui *offset* “ne-aliniat”):

i) prima variantă abordează modul clasic de tratare, folosit până acum: afișarea unui mesaj de eroare și terminarea execuției programului ;

ii) a doua variantă ilustrează un nou mod de tratare: “corectarea” prin program a datelor de intrare “invalide” introduse de utilizator și continuarea execuției programului cu aceste date “corectate” .



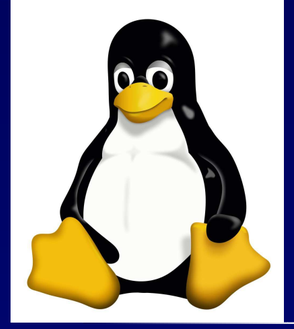
Exemplul #2: O mapare “partajată”, cu permisiuni *read&write*

Aici se ilustrează folosirea apelului `mmap` pentru realizarea unei mapări “partajate”, cu permisiuni de acces *read & write*, a unei porțiuni specificate dintr-un fișier.

A se vedea programul `mmap_ex2f.c` ([1]).

Observație: pentru explicații mai detaliate ale acestui program și descrierea comportamentului său la execuție, consultați exemplul [SecondDemo – mmap_ex2] prezentat în suportul de laborator #9.

Acest program demonstrează *citiri și scrieri direct în memorie* a informației mapate din fișier, în locul utilizării interfeței clasice de acces I/O la disc (adică fără a folosi apelurile de sistem `read` și `write`, sau funcții de I/O din biblioteci precum `libc`), fiind obținut prin adăugarea și a unor operații de “scriere” la programul din exemplul precedent, plus toate modificările necesare în acest scop.



Exemplul #3: O mapare “partajată”, cu scrieri “înafara” mapării

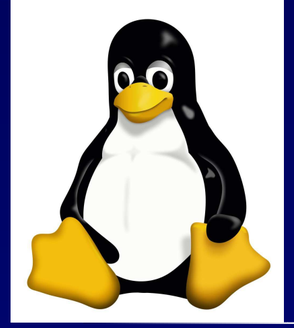
Un exemplu ce ilustrează folosirea apelului `mmap` pentru realizarea unei mapări de tip “partajată”, cu permisiuni de acces *read & write*, a unei porțiuni specificate dintr-un fișier, și care în plus ilustrează ce se întâmplă când scriem la adrese situate “înafara” mapării respective (*i.e.*, la adrese de memorie situate după cea corespunzătoare sfârșitului porțiunii de fișier mapate în memorie).

A se vedea variantele de program `mmap_ex3a.c` și `mmap_ex3b.c` ([1]).

Observație: pentru explicații mai detaliate ale acestor programe și descrierea comportamentului lor la execuție, consultați exemplul [ThirdDemo – mmap_ex3] prezentat în suportul de laborator #9.

Acest program demonstrează cazul scrierilor “înafara” mapării respective, precum și efectul lor asupra fișierului de pe disc (*i.e.*, “Are loc actualizarea modificărilor în fișierul de pe disc sau nu?”), fiind obținut prin adăugarea, la programul din exemplul precedent, de noi operații de “scriere” la adrese de memorie situate după cea corespunzătoare sfârșitului porțiunii de fișier mapate în memorie.

Cele două variante de program tratează două cazuri diferite: scrieri la adrese situate “înafara” mapării respective, dar totuși în interiorul ultimei pagini alocate mapării, versus scrieri la adrese situate “dincolo de” ultima pagină alocată mapării.



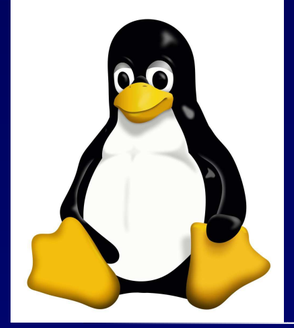
Exemplul #4: O altă mapare “partajată”, pentru crearea unui fișier

Un exemplu ce ilustrează folosirea apelului `mmap` pentru realizarea unei mapări de tip “partajată”, cu permisiuni de acces *read & write*, a unei porțiuni specificate dintr-un fișier, și în care facem doar scrieri în fișier, și nu actualizări de tipul citire + scriere.

A se vedea programul `mmap_ex4c.c` ([1]).

Observație: pentru explicații mai detaliate ale acestui program și descrierea comportamentului său la execuție, consultați exemplul [FourthDemo – mmap_ex4] prezentat în suportul de laborator #9.

Acest program demonstrează doar operații de scriere (fără citire prealabilă), direct în memorie, a conținutului nou pentru acel fișier, urmată de observarea salvării în fișierul de pe disc a informațiilor scrise în memorie. Practic, urmărim să creăm fișierul, cu un anumit conținut (nou) ; nu ne interesează conținutul vechi, în caz că acel fișier exista dinainte.



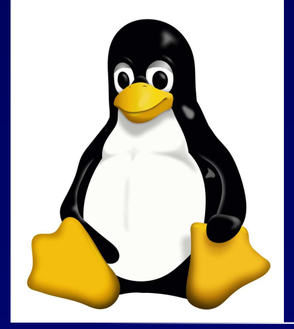
Exemplul #5: O mapare ne-persistentă cu nume

Un exemplu ce ilustrează folosirea apelului `shm_open` pentru crearea unui obiect de memorie partajată și a apelului `mmap` pentru realizarea unei mapări ne-persistente cu nume a acelui obiect.

A se vedea perechea de programe cooperante `shm_producer.c` și `shm_consumer.c` ([1]).

Observație: pentru explicații mai detaliate ale acestor programe și descrierea comportamentului lor la execuție, consultați exemplul [FifthDemo – shm_producer + shm_consumer] prezentat în suportul de laborator #9.

În concluzie, acest exemplu ilustrează modul de creare a unei **mapări ne-persistente cu nume** (*i.e.*, a unui obiect de tipul POSIX *shared memory object*), în scopul utilizării acestui obiect pentru realizarea unei comunicații între două procese cooperante, unul cu rol de producător și celălalt cu rol de consumator.



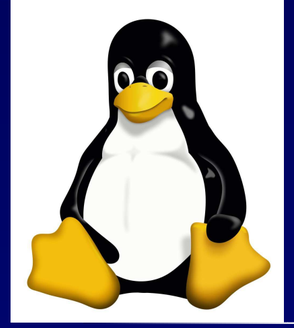
Exemplul #6: O mapare ne-persistentă anonimă

Un exemplu ce ilustrează folosirea apelului `mmap` pentru realizarea unei mapări ne-persistente anonime.
În plus, se mai ilustrează ce se întâmplă la încercarea de accesare a unor adrese virtuale situate în paginile ce urmează, în spațiul virtual de adresare al procesului, “după” o mapare (nu neapărat anonimă) – fie accese de citire (în prima variantă de program), fie accese de scriere (în a doua variantă).

A se vedea cele două variante `anon-mmap_ex1.c` și `anon-mmap_ex2.c` ([1]).

Observație: pentru explicații mai detaliate ale acestor două versiuni de program și descrierea comportamentului lor la execuție, consultați exemplul [SixthDemo – anon-mmap_ex{1,2}] prezentat în suportul de laborator #9.

În concluzie, acest exemplu ilustrează modul de creare a unei **mapări ne-persistente anonime**, iar în lecția practică următoare vom vedea cum utilizăm o mapare anonimă pentru realizarea unei comunicații între două procese cooperante, înrudite prin apelul `fork`.



Alte exemple de programe cu mapări

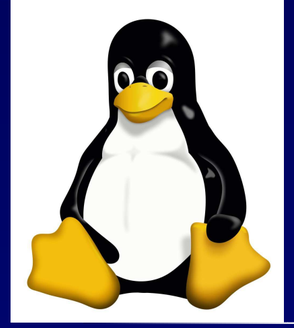
Demo: exercițiul rezolvat [txt2bin_write-mapped-file] , prezentat în suportul online de laborator ([2]), ilustrează un exemplu de program care citește de la tastatură o secvență de numere întregi, introduse prin reprezentarea lor textuală, și le scrie în memorie (deci în format binar), în maparea corespunzătoare fișierului de ieșire specificat.

Notă: practic, acest program este o reimplementare, utilizând o mapare în locul funcțiilor clasice de I/O, a programului demonstrativ [txt2bin_write-file] , prezentat în suportul online al laboratorului #7.

* * *

Demo: exercițiul rezolvat [bin2txt_read-mapped-file] , prezentat în suportul online de laborator ([2]), ilustrează un exemplu de program care afișează pe ecran reprezentarea textuală a numerelor citite prin inițializarea mapării în memorie a unui fișier de date specificat de pe disc, fișier ce conține o secvență numere stocate în format binar.

Notă: practic, acest program este o reimplementare, utilizând o mapare în locul funcțiilor clasice de I/O, a programului demonstrativ [bin2txt_read-file] , prezentat în suportul online al laboratorului #7.



Alte exemple de programe cu mapări (cont.)

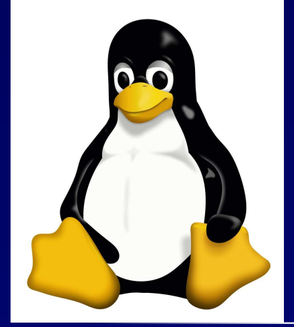
Demo: exercițiul rezolvat [Demo ’data race’_shmem #1 : . . .], prezentat în suportul online de laborator ([2]), ilustrează șablonul de cooperare Producător-Consumator ce a fost prezentat în cursul teoretic #6, particularizat pe un exemplu concret de informație ce este produsă de un proces și consumată de celălalt proces. Se utilizează un fișier mapat în memoria ambelor programe pentru a obține o zonă de memorie partajată prin intermediul căreia se transmite informația de la procesul producător la cel consumator și, în plus, NU se folosește niciun mecanism de sincronizare a citirilor și scrierilor în regiunea de memorie partajată, ceea ce are ca posibil efect citiri de informații “incorecte”.

* * *

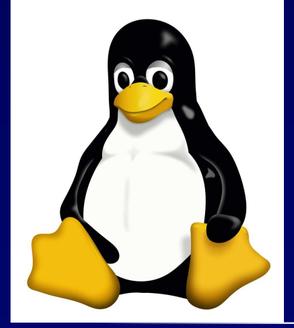
Demo: exercițiul rezolvat [Demo ’data race’_shmem #2 : . . .], prezentat în suportul online de laborator ([2]), ilustrează un alt caz particularizat al șablonului general de cooperare Producător-Consumator, și anume cazul *buffer*ului cu capacitatea 1. La fel ca la primul exemplu, nici aici NU se folosește niciun mecanism de sincronizare a citirilor și scrierilor în regiunea de memorie partajată, ceea ce are ca posibil efect citiri “incorecte” a informațiilor.

* * *

Important: aceste două exemple mai ilustrează în plus și fenomenul de *data race* ce a fost prezentat la începutul cursului teoretic #5, având rolul de a vă atrage atenția asupra nevoii de folosire a unor tehnici specifice pentru sincronizarea execuției programelor, în scopul “reparării” programelor ca să nu (mai) “sufere” de acest fenomen nedorit.



Agenda



Modele de comunicație între procese (IPC)

(a) Modelul de comunicație prin memorie partajată (“*shared-memory model*”)

— prin fișiere mapate în memorie, sau mapări ne-persistente cu nume și anonime, ș.a.

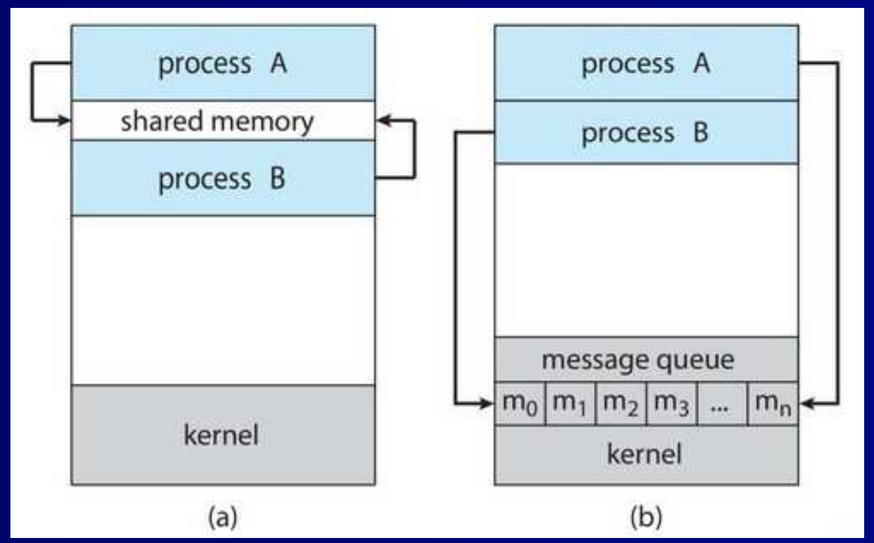
(b) Modelul de comunicație prin schimb de mesaje (“*message-passing model*”)

— *comunicație locală*, prin:

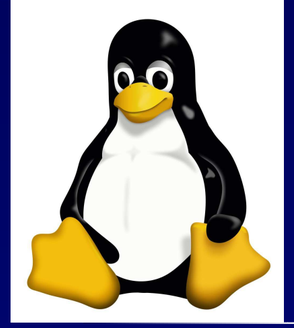
- canale de comunicație cu nume și anonime
- cozi de mesaje

— *comunicație la distanță*, prin:

- *socket*-uri (+ protocoale de comunicație)



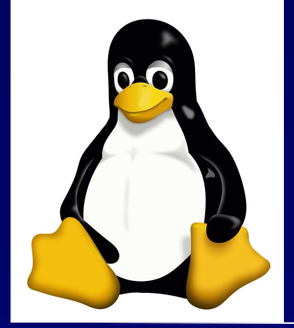
(a) Caracteristici ale modelului IPC prin memorie partajată:
i) Comunicarea se realizează sub controlul proceselor utilizatorilor, nu al sistemului de operare ; ii) SO-ul trebuie să ofere mecanisme care să permită proceselor utilizatorilor să-și sincronizeze acțiunile atunci când accesează memoria partajată – *e.g.* de obicei, procesele își sincronizează accesele la o regiune de memorie partajată folosind semafoare POSIX .
(b) Caracteristici ale modelului IPC prin schimb de mesaje:
i) SO-ul oferă un *sistem de transmitere a mesajelor*: procesele comunică între ele fără a recurge la variabile partajate, prin utilizarea a două tipuri de operații: send(*mesaj*) și receive(*mesaj*) .



Șabloane arhitecturale de cooperare și sincronizare

- Șabloanele arhitecturale studiate în cursurile teoretice #5 și #6 : ’Critical Section’, ’Producer-Consumer’, ’Readers and Writers (CREW)’, ș.a.
- Șablonul de cooperare ’Supervisor / workers’ (aka ’Master / slaves’) :
  Este un șablon de calcul paralel aplicabil atunci când avem o problemă complexă a cărei rezolvare se poate “diviza” în mai multe sub-probleme **ce pot fi apoi rezolvate, în paralel, independent una de alta**, iar la final rezultatele parțiale obținute pot fi “agregate” pentru a obține rezultatul final al problemei inițiale.
- Șablonul de sincronizare ’Token ring’ :
  Este un șablon de sincronizare care surprinde următoarea situație: avem un număr oarecare p de procese, fiecare având de executat, în mod repetitiv, câte o acțiune specifică A_i , cu $i = 1, ..., p$, și se cere sincronizarea execuției lor în paralel, astfel încât ordinea de execuție (*i.e.*, *trace*-ul execuției) să fie precis următoarea: $A_1, A_2, \ldots, A_p$, repetată de un anumit număr de ori.
- Șablonul arhitectural ’Client / server’ :
  Este un șablon de calcul paralel / distribuit aplicabil atunci când avem un *furnizor al unui serviciu* (numit “Server”), care oferă un anumit “serviciu de calcul”, și o mulțime oarecare de *consumatori ai acelui serviciu* (numiți “Clienți”), care vor accesa posibil **simultan** “serviciul” oferit de acel furnizor.

Pentru detalii suplimentare despre aceste șabloane, puteți consulta materialul disponibil aici.



Semafoare POSIX

API-ul POSIX pentru semafoare ([6]) permite proceselor (și firelor de execuție) să-și sincronizeze acțiunile.

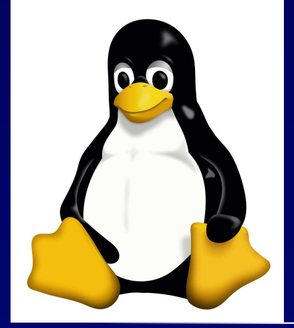
Un *semafor* este un număr întreg a cărui valoare nu este niciodată permis să scadă sub zero.
Două operații pot fi efectuate pe semafoare: incrementarea valorii semaforului cu unu (`sem_post`) și decrementarea valorii semaforului cu unu (`sem_wait`). Dacă valoarea unui semafor este zero, atunci o operație `sem_wait` se va bloca până când valoarea semaforului devine mai mare decât zero.

Semafoarele POSIX sunt de două feluri :

- Semafoare fără nume (*i.e.*, anonime) : acestea pot fi partajate de mai multe procese prin plasarea lor într-o regiune de memorie partajată de acele procese (*e.g.*, un obiect POSIX de memorie partajată creat folosind `shm_open`) ; respectiv, pot fi partajate de mai multe fire de execuție dintr-un proces prin plasarea lor într-o regiune de memorie partajată de acele fire de execuție (*e.g.*, o variabilă globală în programul *multi-threaded* respectiv).

  Operații ce pot fi efectuate pe semafoarele anonime ([6]) : i) Înainte de a fi utilizat, un semafor fără nume trebuie inițializat folosind `sem_init`. ii) Apoi poate fi utilizat folosind operațiile `sem_post` și `sem_wait`. iii) Iar când semaforul nu mai este necesar și înainte ca memoria în care este localizat semaforul să fie dealocată, semaforul ar trebui să fie distrus folosind `sem_destroy`.

- Semafoare cu nume : a se vedea următorul *slide*.



Semafoare POSIX (cont.)

Semafoare cu nume :

Un semafor cu nume este identificat printr-un nume de forma */un-nume* . Două procese pot opera pe același semafor cu nume, prin specificarea aceluiași nume în apelul `sem_open`.

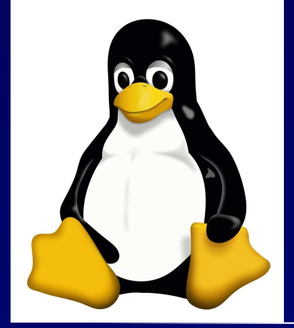
Operații ce pot fi efectuate pe semafoarele cu nume ([6]) :

- `sem_open` : creează un nou semafor cu nume, sau deschide un semafor cu nume deja existent. Returnează adresa noului semafor, adresă ce este utilizată în celelalte funcții enumerate mai jos.
- `sem_post` / `sem_wait` : cele două operații ce pot fi efectuate pe semafoare, descrise mai sus.
- `sem_close` : închide semaforul cu nume, atunci când procesului apelant nu-i mai este necesar.
- `sem_unlink` : elimină din sistem un semafor cu nume.

Persistență : semafoarele POSIX cu nume au *persistență la nivel de kernel* – după creare, un semafor cu nume va exista până când sistemul este oprit / repornit, sau până când va fi șters cu `sem_unlink`.

Accesarea semafoarelor cu nume prin sistemul de fișiere : în `Linux`, semafoarele cu nume sunt create într-un sistem de fișiere virtual de tip `tmpfs`, montat de obicei sub `/dev/shm`, cu nume de fișiere de forma `sem.`*`un-nume`* .

Demo: exercițiul rezolvat [MyCritSec #general : . . .] , prezentat în suportul online de laborator ([2]), ilustrează folosirea semafoarelor pentru asigurarea excluderii mutuale în șablonul general de sincronizare ’Problema Secțiunii Critice’, ce a fost prezentat în cursul teoretic #5.



Bibliografie obligatorie

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

- https://edu.info.uaic.ro/sisteme-de-operare/SO/lectures/Linux/demo/mmap/

[2] Suportul de laborator online asociat acestei prezentări:

- https://edu.info.uaic.ro/sisteme-de-operare/SO/support-lessons/C/suport_lab9.html

Bibliografie suplimentară:

[3] Cap. 49, 50, 51, 53 și 54 din cartea “The Linux Programming Interface : A Linux and UNIX System Programming Handbook”, autor M. Kerrisk, editura No Starch Press, 2010.

- https://edu.info.uaic.ro/sisteme-de-operare/SO/books/TLPI1.pdf

[4] POSIX API for file mappings : man 2 mmap / munmap , man 2 msync , man 2 mprotect , man 2 mincore , man 2 mlock / munlock , etc.

[5] POSIX API for shared memory : man 7 shm_overview , man 3 shm_open , etc.

[6] POSIX API for semaphores : man 7 sem_overview , man 3 sem_post , man 3 sem_wait ; only for named semaphores: man 3 sem_open , man 3 sem_close , man 3 sem_unlink ; and only for unnamed semaphores: man 3 sem_init , man 3 sem_destroy .