Laborator #7 : exerciții de laborator

Sumar:

I) Exerciții de programare a unor probleme de sincronizare cu fișiere -- diverse procesări ce utilizează lacăte pentru acces exclusiv :

***Observație*:** pentru ușurința de programare și pentru eficiența operațiilor I/O cu discul, în exercițiile ce urmează în acest laborator, vă recomand ca tipurile de date numerice să le stocați în fișiere folosind reprezentarea *binară* a lor, care are avantajul că stocarea acestor reprezentări ocupă un număr fix de octeți! În acest context, reprezentarea *textuală* a tipurilor de date numerice nu este cea mai adecvată, și nici cea mai eficientă, deoarece stocarea acestei reprezentări ocupă un număr variabil de octeți în fișier, în funcție de valoarea acelui număr!

Întrebare: La ce sunt utile lacătele în citire?

Răspuns: Lacătele în citire pot fi deținute simultan de mai multe procese pe o anumită porțiune de fișier, dar NU în același timp cu un lacăt în scriere pe aceeași porțiune de fișier! Deci putem face citiri simultane SAU o singură scriere, conform șablonului de cooperare CREW (concurrent-read-exclusive-write) descris în cursul teoretic #6. (A se revedea, în acest sens, și observația de la slide-ul 12/18 din lecția practică despre lacăte pe fișiere.)
Dacă citirile și scrierile pe care le facem în fișier nu sunt atomice [i.e., o informație de procesat trebuie citită prin mai multe multe apeluri read() (e.g., o înregistrare dintr-o bază de date, formată din mai multe câmpuri, iar fiecare câmp trebuie citit prin câte un apel read() individual), și/sau ceva similar pentru operațiile de scriere], atunci este obligatoriu să implementați mecanismul de sincronizare descris la șablonul de cooperare CREW, folosind lacăte în citire și în scriere !

Notă: în cele de mai jos, veți găsi și exerciții ce se pretează pentru rezolvare cu șablonul de cooperare CREW.

a) *Exerciții propuse spre rezolvare* :

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

Setul #1 | Setul #2

Setul 1

1. [MyCritSec #3]
   Implementați **problema secțiunii critice** prezentată în cursul teoretic #5, în scenariul următor:
   Se consideră, drept resursă partajabilă de mai multe procese, un fișier *binar* cu baza de date folosită pentru gestiunea produselor dintr-un magazin. Înregistrările din acest fișier reprezintă perechi de forma: `(cod_produs,stoc)`, unde `cod_produs` este un număr unic (i.e., apare o singură dată în baza de date) de tipul `int`, iar `stoc` este un număr de tipul `float`, reprezentând cantitatea din acel produs (exprimată în unitatea de măsură specifică pentru acel tip de produs, e.g. kilograme, litri, etc.), disponibilă în acel magazin. Perechile de numere întregi și reale sunt reprezentate *binar*, nu *textual*, în fișierul rescpectiv! Asupra acestei baze de date se vor efectua operațiuni de actualizare a stocurilor de produse, conform celor descrise mai jos.
   Scrieți un program C care să efectueze diverse operații de vânzare/cumpărare de produse, la intervale variate de timp, operațiile fiind specificate, într-un **fișier text** cu instrucțiuni, prin secvențe de forma:
   `cod_produs +cantitate` și/sau `cod_produs -cantitate`,

reprezentând cumpărarea și respectiv vânzarea cantității specificate din produsul având codul `cod_produs` specificat.
Pentru fiecare instrucțiune de cumpărare/vânzare din fișierul de instrucțiuni, programul va căuta în fișierul resursă (i.e., baza de date) specificat înregistrarea cu codul `cod_produs` specificat în instrucțiunea respectivă, iar dacă există o astfel de înregistrare, atunci va actualiza valoarea stocului acelui produs în mod corespunzător, dar NUMAI dacă această operație NU conduce la obținerea unei valori negative pentru stoc, altfel va afișa un mesaj de eroare corespunzător.
Dacă nu există codul `cod_produs` specificat în instrucțiunea respectivă, iar operația propusă este `-cantitate`, programul va afișa un mesaj de eroare corespunzător și se va opri din procesarea fișierului de instrucțiuni. Iar dacă nu există codul `cod_produs` specificat în instrucțiunea respectivă, însă operația propusă este `+cantitate`, atunci programul va adăuga o nouă înregistrare, cu valoarea: `(cod_produs,cantitate)`, în baza de date respectivă.

Cerințe:

- Programul va **accesa fișierul resursă în manieră cooperantă**, folosind lacăte în scriere pe durata de efectuare a fiecărei operațiuni de actualizare a stocului, astfel încât să permită execuția simultană a două sau mai multor instanțe ale programului, fără să apară efecte nedorite datorită fenomenelor de *data race*.
  Cu alte cuvinte, ideea este de a executa simultan mai multe instanțe ale programului, care vor accesa concurent același fișier și îl vor modifica, coordonându-și însă accesul exclusiv (doar) la secțiunile modificate/scrise, astfel încât să nu apară efecte nedorite datorită fenomenelor de *data race*.
  (Indicație #1: pentru eficiența și ușurința de programare, perechile de numere de tipul `int` și `float` vor fi reprezentate *binar* în fișier, și nu *textual*, astfel încât fiecare număr întreg, respectiv real, să ocupe exact `sizeof(int)` octeți, respectiv `sizeof(float)` octeți. În acest fel, toate înregistrările din fișier vor avea lungime fixă (și anume, `sizeof(int)+sizeof(float)` octeți), ceea ce va ușura mult prelucrarea lor.)
- Lacătele se vor pune numai pe porțiunea de fișier strict necesară și numai pe durata minimă necesară (asemănător ca la versiunea 4 a programului demonstrativ `access` prezentat în lecția practică despre lacăte pe fișiere).
  (Indicație #2: se vor folosi apelurile de sistem open(), read(), write(), close() și respectiv fcntl() pentru punerea de lacăte pe porțiunile minimale, strict necesare, din fișierul de lucru, pe care lucrează la un moment dat o instanță a programului, iar blocajele vor fi păstrate doar pe durata minimă de timp necesară.)
- Operațiile de actualizare vor fi implementate astfel încât să afișeze pe ecran mesaje explicative despre ceea ce se execută, fiecare mesaj fiind prefixat de PID-ul procesului ce execută respectiva operație de actualizare (pentru a putea face distincție între procesele, i.e. instanțele de execuție paralelă ale programului, ce afișează câte ceva). De asemenea, se va introduce câte o scurtă *pauză* (sub 1 secundă) între orice două operații de actualizare realizate succesiv de program.
- Se va pregăti un **mediu pentru testare**, compus din: programul executabil, câte un fișier de instrucțiuni pentru fiecare instanță a executabilului lansată în execuție, fișierul asupra căruia se vor opera modificările, precum și un script bash care să lanseze în execuții paralele programul cu parametrii corespunzători (câte un fișier de instrucțiuni), adică o lansare pentru test ar putea fi de forma:

```
UNIX> ./prg-sc3 depozit.bin instr1.txt & ./prg-sc3 depozit.bin instr2.txt & ./prg-sc3 depozit.bin instr3.txt & ...
```

  (Indicație #3: pentru a scrie scriptul bash pomenit mai sus, puteți să luați scriptul de la exercițiul rezolvat [Run SPMD programs] din Laboratorul #5 și să-l modificați astfel încât să poată transmite doi parametri către instanțele jobului SPMD pe care-l creează: primul parametru ar fi același pentru toate instanțele, i.e. numele fișierului cu stocuri, iar al doilea parametru ar fi specific pentru fiecare instanță, i.e. numele fișierului de instrucțiuni pentru instanța respectivă. Astfel adaptat, scriptul va putea fi invocat printr-o linie de comandă de forma:

```
UNIX> ./RunMySPMD.sh ./prg-sc3 10 depozit.bin instr1.txt instr2.txt ... instr10.txt
```

  Evident, în loc de 10, se va putea apela cu un număr întreg pozitiv oarecare.)

Show / Hide some suggestions for solving this problem

Ideea de rezolvare:
Mai întâi, (re)citiți cu atenție observațiile generale și sugestiile de rezolvare date la problema rezolvată [MyCritSec #1] din suportul de laborator!

- **Pasul 1**: Scrieți mai întâi o variantă a programului cerut, **fără** folosirea de lacăte pentru acces exclusiv la secțiunea critică din program, și experimentați cu această variantă, astfel:
  i) întâi depanați programul în manieră secvențială (i.e., executați o singură instanță a programului), pentru a elimina eventualele bug-uri de natură secvențială!
  ii) după ce ați corectat toate bug-urile de natură secvențială, treceți la testarea programului în context de execuție paralelă și concurentă conform șablonului SPMD, i.e. executați, de multe ori, programul sub formă de joburi SPMD, ca să observați efectul fenomenelor de *data race*, i.e. obținerea de rezultate inconsistente (incorecte).

- **Pasul 2**: Apoi adăugați, în programul scris la Pasul 1, folosirea de lacăte pentru acces exclusiv la secțiunea critică din program, după ideea din programul `access_v4.c` discutat la curs, i.e. astfel încât să blocați doar porțiunea minimă din fișier și doar pe durata minimă necesară, pentru a asigura excluderea mutuală la execuția secțiunilor critice!
  Experimentați și cu această variantă, în context de execuție paralelă și concurentă conform șablonului SPMD, i.e. executați, de multe ori, programul sub formă de joburi SPMD, ca să observați dacă mai apar rezultate inconsistente (incorecte) datorate fenomenelor de *data race*.
  (*Tip*: nu ar trebui să mai apară! Aceasta, bineînțeles, dacă nu veți face vreo greșeală la implementarea folosirii lacătelor, conform ideilor expuse mai sus.)

1. [MyCritSec #4]
   Implementați **problema secțiunii critice** prezentată în cursul teoretic #5, în scenariul următor:
   Se consideră, drept resursă partajabilă de mai multe procese, un fișier *binar* ce conține perechi de întregi de forma `(cheie,valoare)`, cu proprietatea că valorile de pe prima poziție sunt unice (i.e., fiecare cheie "apare" o singură dată în acel fișier). (Fișierul este *binar* -- adică conține un număr par de întregi scriși *binar* în fișier, nu *textual* !)
   Să se scrie un program C care primește în linia de comandă numele acestui fișier cu perechi de forma `(cheie,valoare)` și acționează asupra fișierului în felul următor: programul va citi dintr-un alt **fișier *binar***, specificat tot ca parametru la linia de comandă, o listă de "comenzi" de forma `(cheie,alter)`, care vor fi tot perechi de întregi, reprezentați în mod *binar*, nu *textual* ! Apoi, programul va parcurge lista comenzilor citite și, pentru fiecare astfel de comandă, va căuta cheia specificată în fișierul partajat. Dacă o găsește, va aduna la valoarea cheii din fișier, valoarea `alter` specificată în comandă, iar dacă nu o găsește, atunci va adăuga la sfârșitul fișierului o nouă cheie `cheia` cu valoarea inițială `alter`.

   Cerințe:

   - Programul va **accesa fișierul partajat în manieră cooperantă**, folosind lacăte în scriere pe durata de efectuare a fiecărei operațiuni de actualizare a valorii asociate unei chei, astfel încât să permită execuția simultană a două sau mai multor instanțe ale programului, fără să apară efecte nedorite datorită fenomenelor de *data race*.
     Cu alte cuvinte, ideea este de a executa simultan mai multe instanțe ale programului, care vor accesa concurent același fișier și îl vor modifica, coordonându-și însă accesul exclusiv (doar) la secțiunile modificate/scrise, astfel încât să nu apară efecte nedorite datorită fenomenelor de *data race*.
     Exemplu: dacă în fișierul partajat, numit de exemplu `perechi.bin`, am inițial o pereche `(2,5)`, iar în două fișiere de comenzi am, printre alte chei, perechile `(2,9)` și respectiv `(2,-1)`, atunci în fișierul de lucru voi găsi perechea `(2,13)`, după încheierea execuției celor două instanțe ale programului, apelate cu cele două fișiere de comenzi.
     (Indicație #1: pentru eficiența și ușurința de programare, numerele întregi vor fi reprezentate *binar* în fișier, și nu *textual*, astfel încât fiecare număr întreg să ocupe exact `sizeof(int)` octeți. În acest fel toate înregistrările din fișier vor avea lungime fixă (și anume, `2*sizeof(int)` octeți), ceea ce va ușura mult prelucrarea lor.)
   - Lacătele se vor pune numai pe porțiunea de fișier strict necesară și numai pe durata minimă necesară (asemănător ca la versiunea 4 a programului demonstrativ `access` prezentat în lecția practică despre lacăte pe fișiere).
     (Indicație #2: se vor folosi apelurile de sistem open(), read(), write(), close() și respectiv fcntl() pentru punerea de lacăte pe porțiunile minimale, strict necesare, din fișierul de lucru, pe care lucrează la un moment dat o instanță a programului, iar blocajele vor fi păstrate doar pe durata minimă de timp necesară.)
   - Operațiile de actualizare vor fi implementate astfel încât să afișeze pe ecran mesaje explicative despre ceea ce se execută, fiecare mesaj fiind prefixat de PID-ul procesului ce execută respectiva operație de actualizare (pentru a putea face distincție între procesele, i.e. instanțele de execuție paralelă ale programului, ce afișează câte ceva). De asemenea, se va introduce câte o scurtă *pauză* (sub 1 secundă) între orice două operații de actualizare realizate succesiv de program.
   - Se va pregăti un **mediu pentru testare**, compus din: programul executabil, câte un fișier de comenzi pentru fiecare instanță a executabilului lansată în execuție, fișierul asupra căruia se vor opera modificările, precum și un script bash care să lanseze în execuții paralele programul cu parametrii corespunzători (câte un fișier de comenzi), adică o lansare pentru test ar putea fi de forma:
     ```
     UNIX> ./prg-sc4 perechi.bin comenzi1.bin & ./prg-sc4 perechi.bin comenzi2.bin & ./prg-sc4 perechi.bin comenzi3.bin & ...
     ```
     (Indicație #3: pentru a scrie scriptul bash pomenit mai sus, puteți să luați scriptul de la exercițiul rezolvat [Run SPMD programs] din Laboratorul #5 și să-l modificați astfel încât să poată transmite doi parametri către instanțele jobului SPMD pe care-l creează: primul parametru ar fi acelasi pentru toate instanțele, i.e. numele fișierului cu perechi, iar al doilea parametru ar fi specific pentru fiecare instanță, i.e. numele fișierului cu lista de comenzi pentru instanța respectivă. Astfel adaptat, scriptul va putea fi invocat printr-o linie de comandă de forma:
     ```
     UNIX> ./RunMySPMD.sh ./prg-sc4 5 perechi.bin comenzi1.bin comenzi2.bin ... comenzi5.bin
     ```
     Evident, în loc de 5, se va putea apela cu un număr întreg pozitiv oarecare.)

Show / Hide some suggestions for solving this problem

Ideea de rezolvare:
Mai întâi, (re)citiți cu atenție observațiile generale și sugestiile de rezolvare date la problema rezolvată [MyCritSec #1] din suportul de laborator!

- **Pasul 1**: Scrieți mai întâi o variantă a programului cerut, **fără** folosirea de lacăte pentru acces exclusiv la secțiunea critică din program, și experimentați cu această variantă, astfel:
  i) întâi depanați programul în manieră secvențială (i.e., executați o singură instanță a programului), pentru a elimina eventualele bug-uri de natură secvențială!
  ii) după ce ați corectat toate bug-urile de natură secvențială, treceți la testarea programului în context de execuție paralelă și concurentă conform șablonului SPMD, i.e. executați, de multe ori, programul sub formă de joburi SPMD, ca să observați efectul fenomenelor de *data race*, i.e. obținerea de rezultate inconsistente (incorecte).

- **Pasul 2**: Apoi adăugați, în programul scris la Pasul 1, folosirea de lacăte pentru acces exclusiv la secțiunea critică din program, după ideea din programul `access_v4.c` discutat la curs, i.e. astfel încât să blocați doar porțiunea minimă din fișier și doar pe durata minimă necesară, pentru a asigura excluderea mutuală la execuția secțiunilor

critice!
Experimentați și cu această variantă, în context de execuție paralelă și concurentă conform șablonului SPMD, i.e. executați, de multe ori, programul sub formă de joburi SPMD, ca să observați dacă mai apar rezultate inconsistente (incorecte) datorate fenomenelor de *data race*.
(*Tip*: nu ar trebui să mai apară! Aceasta, bineînțeles, dacă nu veți face vreo greșeală la implementarea folosirii lacătelor, conform ideilor expuse mai sus.)

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

Iată alte câteva exerciții de programare C cu sincronizări pe fișiere (folosind lacăte pentru acces exclusiv), pe care să încercați să le rezolvați singuri în timpul liber, pentru a vă auto-evalua cunoștințele dobândite în urma acestui laborator:

1. [MyCREW #1] Enunțul problemei

Implementați **problema de sincronizare/șablonul de cooperare CREW** prezentată în cursul teoretic #6, în scenariul următor:
Se consideră, drept resursă partajabilă de mai multe procese, un fișier *binar* cu baza de date folosită pentru gestiunea evidenței studenților înmatriculați la o facultate. Fiecare înregistrare din baza de date va avea forma **`(id_stud, nume_student, an_studiu, stare_înmatriculare, la_buget)`**, unde "id_stud" va fi un număr unic (i.e., apare o singură dată în baza de date) și va fi de tipul int, "nume_student" va fi de tipul string[50] și va reprezenta numele complet al studentului (stocat pe 50 de caractere), "an_studiu" va fi anul de studiu în care este înscris studentul și va fi de tipul int, "stare_înmatriculare" va fi de tipul boolean și semnifică starea curentă, înmatriculat sau exmatriculat, a studentului respectiv, iar "la_buget" va fi de tipul int boolean și semnifică tipul de finanțare, de la buget sau cu taxă, a studentului respectiv.
Notă: eventual, puteți să mai adăugați și alte câmpuri de informații, în formatul înregistrărilor din baza de date, pe lângă cele deja descrise mai sus.
Datele din înregistrările bazei de date vor fi reprezentate în mod *binar*, nu *textual*, în fișierul respectiv!
Asupra acestei baze de date se vor efectua operațiuni de adăugare/actualizare de informații despre studenți și, respectiv, acțiuni de căutare de informații despre studenți.
Să se scrie un program C care să efectueze diverse operații de adăugare/actualizare de date și de căutare de date, la intervale variate de timp, operațiile fiind specificate într-un fișier text cu instrucțiuni (ce va fi transmis programului la linia de comandă), prin secvențe de forma:

- `adauga` *`nume_student an_studiu stare_înmatriculare la_buget`* -- pentru a adăuga o intrare nouă în baza de date;
- `modifica` *`id_stud an_studiu stare_înmatriculare la_buget`* -- pentru a modifica o intrare existentă în baza de date;
- `cauta-id` *`id_stud`* -- pentru a afișa informațiile dintr-o intrare existentă în baza de date, specificată prin id-ul său;
- `cauta-nume` *`nume_student`* -- pentru a afișa informațiile dintr-o intrare existentă în baza de date, specificată prin numele studentului.

Pentru instrucțiunea `adauga`, programul va calcula cea mai mică valoare "id_stud" ce nu se află în fișierul resursă și va insera intrarea nouă cu această valoare "id_stud" și cu celelalte valori specificate în instrucțiune.
Pentru instrucțiunea `modifica`, programul va căuta intrarea cu id-ul "id_stud" primit în argumentele instrucțiunii respective și va înlocui datele din fișierul resursă cu cele primite ca argumente ale acelei instrucțiuni. În cazul în care "id_stud" nu este prezent în baza de date, se va afișa un mesaj de eroare corespunzător și se va opri procesarea fișierului de instrucțiuni.
Pentru instrucțiunea `cauta-id`, programul va căuta intrarea cu id-ul "id_stud" specificat ca argument al instrucțiunii respective și va afișa datele din fișierul resursă găsite în înregistrarea cu acel id. În cazul în care "id_stud" nu este prezent în baza de date, se va afișa un mesaj de eroare corespunzător și se va opri procesarea fișierului de instrucțiuni.
Pentru instrucțiunea `cauta-nume`, programul va căuta intrarea, sau intrările, ce conțin valoarea "nume_student" specificată ca argument al instrucțiunii respective și va afișa datele din fișierul resursă găsite în înregistrarea, sau inregistrările, cu acel nume. În cazul în care nu există nicio înregistrare cu acel nume în baza de date, se va afișa un mesaj de eroare corespunzător și se va opri procesarea fișierului de instrucțiuni.

Cerințe:

- Programul va **accesa fișierul resursă în manieră cooperantă**, folosind lacăte în scriere pe durata de efectuare a fiecărei operațiuni de actualizare a bazei de date (i.e., adăugare sau modificare), respectiv folosind lacăte în citire pe durata de efectuare a fiecărei operațiuni de consultare a bazei de date (i.e., căutare după ID sau după nume) astfel încât să permită execuția simultană a două sau mai multor instanțe ale programului, fără să apară efecte nedorite datorită fenomenelor de *data race*.
  Cu alte cuvinte, ideea este de a executa simultan mai multe instanțe ale programului, care vor accesa concurent același fișier și îl vor modifica sau consulta, coordonându-și însă accesul, exclusiv în scriere și concurent în citire, (doar) la secțiunile scrise/citite, astfel încât să nu apară efecte nedorite datorită fenomenelor de *data race*.
  (Indicație #1: pentru eficiența și ușurința de programare, valorile specificate în fiecare înregistrare a bazei de date, vor fi reprezentate *binar* în fișier, și nu *textual*, astfel încât fiecare valoare să ocupe exact `2*sizeof(int)+2*sizeof(boolean)+50` octeți În acest fel, toate înregistrările din fișier vor avea lungime fixă, ceea ce va ușura mult prelucrarea lor.)

- Lacătele, în citire și în scriere, se vor pune numai pe porțiunea de fișier strict necesară și numai pe durata minimă necesară (asemănător ca la versiunea 4 a programului demonstrativ `access` prezentat în lecția practică despre lacăte pe fișiere).
  (Indicație #2: se vor folosi apelurile de sistem open(), read(), write(), close() și respectiv fcntl() pentru punerea de lacăte pe porțiunile minimale, strict necesare, din fișierul de lucru, pe care lucrează la un moment dat o instanță a programului, iar blocajele vor fi păstrate doar pe durata minimă de timp necesară.)
- Operațiile de actualizare și de consultare vor fi implementate astfel încât să afișeze pe ecran mesaje explicative despre ceea ce se execută, fiecare mesaj fiind prefixat de PID-ul procesului ce execută respectiva operație de actualizare / consultare (pentru a putea face distincție între procesele, i.e. instanțele de execuție paralelă ale programului, ce afișează câte ceva). De asemenea, se va introduce câte o scurtă *pauză* (sub 1 secundă) între orice două operații de actualizare / consultare realizate succesiv de program.
- Se va pregăti un **mediu pentru testare**, compus din: programul executabil, câte un fișier de instrucțiuni pentru fiecare instanță a executabilului lansată în execuție, fișierul asupra căruia se vor opera modificările, precum și un script bash care să lanseze în execuții paralele programul cu parametrii corespunzători (câte un fișier de instrucțiuni), adică o lansare pentru test ar putea fi de forma:
  `UNIX> ./prg-crew1 studs.bin instr1.txt & ./prg-crew1 studs.bin instr2.txt & ./prg-crew1 studs.bin instr3.txt & ...`
  (Indicație #3: pentru a scrie scriptul bash pomenit mai sus, puteți să luați scriptul de la exercițiul rezolvat [Run SPMD programs] din Laboratorul #5 și să-l modificați astfel încât să poată transmite doi parametri către instanțele jobului SPMD pe care-l creează: primul parametru ar fi acelasi pentru toate instanțele, i.e. numele fișierului cu baza de date, iar al doilea parametru ar fi specific pentru fiecare instanță, i.e. numele fișierului de instrucțiuni pentru instanța respectivă. Astfel adaptat, scriptul va putea fi invocat printr-o linie de comandă de forma:
  `UNIX> ./RunMySPMD.sh ./prg-crew1 5 studs.bin instr1.txt instr2.txt ... instr5.txt`
  Evident, în loc de 5, se va putea apela cu un număr întreg pozitiv oarecare.)

Show / Hide some suggestions for solving this problem

Ideea de rezolvare:
Mai întâi, (re)citiți cu atenție observațiile generale și sugestiile de rezolvare date la problema rezolvată [MyCritSec #1] din suportul de laborator!

- **Pasul 1**: Scrieți mai întâi o variantă a programului cerut, **fără** folosirea de lacăte pentru acces CREW la secțiunile critice din program, și experimentați cu această variantă, astfel:
  i) întâi depanați programul în manieră secvențială (i.e., executați o singură instanță a programului), pentru a elimina eventualele bug-uri de natură secvențială!
  ii) după ce ați corectat toate bug-urile de natură secvențială, treceți la testarea programului în context de execuție paralelă și concurentă conform șablonului SPMD, i.e. executați, de multe ori, programul sub formă de joburi SPMD, ca să observați efectul fenomenelor de *data race*, i.e. obținerea de rezultate inconsistente (incorecte).

- **Pasul 2**: Apoi adăugați, în programul scris la Pasul 1, folosirea de lacăte pentru acces CREW la secțiunile critice din program, după ideea din programul `access_v4.c` discutat la curs, i.e. astfel încât să blocați doar porțiunea minimă din fișier și doar pe durata minimă necesară, pentru a asigura excluderea mutuală la execuția secțiunilor critice!
  Experimentați și cu această variantă, în context de execuție paralelă și concurentă conform șablonului SPMD, i.e. executați, de multe ori, programul sub formă de joburi SPMD, ca să observați dacă mai apar rezultate inconsistente (incorecte) datorate fenomenelor de *data race*.
  (*Tip*: nu ar trebui să mai apară! Aceasta, bineînțeles, dacă nu veți face vreo greșeală la implementarea folosirii lacătelor, conform ideilor expuse mai sus.)

2. [MyCREW #2] Enunțul problemei

Implementați **problema de sincronizare/șablonul de cooperare CREW** prezentată în cursul teoretic #6, în scenariul următor:
Se consideră, drept resursă partajabilă de mai multe procese, un fișier *binar* cu baza de date folosită pentru auditarea zborurilor avioanelor dintr-un anumit aeroport. Fiecare înregistrare din baza de date va avea forma **`(id_zbor, directie, ora_plecare, ora_sosire)`**, unde "id_zbor" va fi un număr unic (i.e., apare o singură dată în baza de date) și va fi de tipul int, "directie" va fi de tipul `boolean/int/char` și va reprezenta directia înspre care se îndreapta avionul (i.e., ingoing sau outgoing), iar "ora_plecare" și "ora_sosire" vor fi de tipul `int` și vor reprezenta *timestamp*-ul (i.e., numărul de secunde scurse de la data de 1.1.1970 00:00 și până în prezent; dar pentru a simplifica, puteți alege orice dată sau oră doriți ca și "origine") la care avionul respectiv va decola, respectiv la care va ateriza pe acel aeroport.
Notă: eventual, puteți să mai adăugați și alte câmpuri de informații, în formatul înregistrărilor din baza de date, pe lângă cele deja descrise mai sus.
Datele din înregistrările bazei de date vor fi reprezentate în mod *binar*, nu *textual*, în fișierul respectiv!
Asupra acestei baze de date se vor efectua operațiuni de adăugare/actualizare zboruri și, respectiv, acțiuni de căutare de informații despre zboruri.
Să se scrie un program C care să efectueze diverse operații de adăugare/actualizare și de căutare de zboruri, la intervale variate de timp, operațiile fiind specificate într-un fișier text cu instrucțiuni (ce va fi transmis programului la linia de comandă), prin secvențe de forma:

- `adauga` *`directie ora_plecare ora_sosire`* -- pentru a adăuga o intrare nouă în baza de date;
- `modifica` *`id_zbor directie ora_plecare ora_sosire`* -- pentru a modifica o intrare existentă în baza de date;
- `cauta` *`id_zbor`* -- pentru a afișa informațiile dintr-o intrare existentă în baza de date, specificată prin id-ul său.

Pentru instrucțiunea `adauga`, programul va calcula cea mai mică valoare "id_zbor" ce nu se află în fișierul resursă și va insera intrarea nouă cu această valoare "id_zbor" și cu celelalte valori specificate în instrucțiune.
Pentru instrucțiunea `modifica`, programul va căuta intrarea cu id-ul "id_zbor" primit ca argument al instrucțiunii respective și va înlocui datele din fișierul resursă cu cele primite ca argumente ale acelei instrucțiuni. În cazul în care "id_zbor" nu este prezent în baza de date, se va afișa un mesaj de eroare corespunzător și se va opri procesarea fișierului de instrucțiuni.
Pentru instrucțiunea `cauta`, programul va căuta intrarea cu id-ul "id_zbor" primit ca argument al instrucțiunii respective și va afișa datele din fișierul resursă găsite în înregistrarea cu acel id. În cazul în care "id_zbor" nu este prezent în baza de date, se va afișa un mesaj de eroare corespunzător și se va opri procesarea fișierului de instrucțiuni.

Cerințe:

- Programul va **accesa fișierul resursă în manieră cooperantă**, folosind lacăte în scriere pe durata de efectuare a fiecărei operațiuni de actualizare a bazei de date (i.e., adăugare sau modificare), respectiv folosind lacăte în citire pe durata de efectuare a fiecărei operațiuni de consultare a bazei de date (i.e., căutare după ID sau după nume) astfel încât să permită execuția simultană a două sau mai multor instanțe ale programului, fără să apară efecte nedorite datorită fenomenelor de *data race*.
  Cu alte cuvinte, ideea este de a executa simultan mai multe instanțe ale programului, care vor accesa concurent același fișier și îl vor modifica sau consulta, coordonându-și însă accesul, exclusiv în scriere și concurent în citire, (doar) la secțiunile scrise/citite, astfel încât să nu apară efecte nedorite datorită fenomenelor de *data race*.
  (Indicație #1: pentru eficiența și ușurința de programare, valorile specificate în fiecare înregistrare a bazei de date, vor fi reprezentate *binar* în fișier, și nu *textual*, astfel încât fiecare valoare să ocupe exact `sizeof(`*`tipul-valorii`*`)` octeți În acest fel, toate înregistrările din fișier vor avea lungime fixă, ceea ce va ușura mult prelucrarea lor.)
- Lacătele, în citire și în scriere, se vor pune numai pe porțiunea de fișier strict necesară și numai pe durata minimă necesară (asemănător ca la versiunea 4 a programului demonstrativ `access` prezentat în lecția practică despre lacăte pe fișiere).
  (Indicație #2: se vor folosi apelurile de sistem open(), read(), write(), close() și respectiv fcntl() pentru punerea de lacăte pe porțiunile minimale, strict necesare, din fișierul de lucru, pe care lucrează la un moment dat o instanță a programului, iar blocajele vor fi păstrate doar pe durata minimă de timp necesară.)
- Operațiile de actualizare și de consultare vor fi implementate astfel încât să afișeze pe ecran mesaje explicative despre ceea ce se execută, fiecare mesaj fiind prefixat de PID-ul procesului ce execută respectiva operație de actualizare / consultare (pentru a putea face distincție între procesele, i.e. instanțele de execuție paralelă ale programului, ce afișează câte ceva). De asemenea, se va introduce câte o scurtă *pauză* (sub 1 secundă) între orice două operații de actualizare / consultare realizate succesiv de program.
- Se va pregăti un **mediu pentru testare**, compus din: programul executabil, câte un fișier de instrucțiuni pentru fiecare instanță a executabilului lansată în execuție, fișierul asupra căruia se vor opera modificările, precum și un script bash care să lanseze în execuții paralele programul cu parametrii corespunzători (câte un fișier de instrucțiuni), adică o lansare pentru test ar putea fi de forma:
  ```
  UNIX> ./prg-crew2 aeroport.bin instr1.txt & ./prg-crew2 aeroport.bin instr2.txt & ./prg-crew2 aeroport.bin instr3.txt & ...
  ```
  (Indicație #3: pentru a scrie scriptul bash pomenit mai sus, puteți să luați scriptul de la exercițiul rezolvat [Run SPMD programs] din Laboratorul #5 și să-l modificați astfel încât să poată transmite doi parametri către instanțele jobului SPMD pe care-l creează: primul parametru ar fi acelasi pentru toate instanțele, i.e. numele fișierului cu baza de date, iar al doilea parametru ar fi specific pentru fiecare instanță, i.e. numele fișierului de instrucțiuni pentru instanța respectivă. Astfel adaptat, scriptul va putea fi invocat printr-o linie de comandă de forma:
  ```
  UNIX> ./RunMySPMD.sh ./prg-crew2 5 aeroport.bin instr1.txt instr2.txt ... instr5.txt
  ```
  Evident, în loc de 5, se va putea apela cu un număr întreg pozitiv oarecare.)

Show / Hide some suggestions for solving this problem

Ideea de rezolvare:
Mai întâi, (re)citiți cu atenție observațiile generale și sugestiile de rezolvare date la problema rezolvată [MyCritSec #1] din suportul de laborator!

- **Pasul 1**: Scrieți mai întâi o variantă a programului cerut, **fără** folosirea de lacăte pentru acces CREW la secțiunile critice din program, și experimentați cu această variantă, astfel:
  i) întâi depanați programul în manieră secvențială (i.e., executați o singură instanță a programului), pentru a elimina eventualele bug-uri de natură secvențială!
  ii) după ce ați corectat toate bug-urile de natură secvențială, treceți la testarea programului în context de execuție paralelă și concurentă conform șablonului SPMD, i.e. executați, de multe ori, programul sub formă de joburi SPMD, ca să observați efectul fenomenelor de *data race*, i.e. obținerea de rezultate inconsistente (incorecte).

- **Pasul 2**: Apoi adăugați, în programul scris la Pasul 1, folosirea de lacăte pentru acces CREW la secțiunile critice din program, după ideea din programul `access_v4.c` discutat la curs, i.e. astfel încât să blocați doar porțiunea minimă din fișier și doar pe durata minimă necesară, pentru a asigura excluderea mutuală la execuția secțiunilor critice!
  Experimentați și cu această variantă, în context de execuție paralelă și concurentă conform șablonului SPMD, i.e. executați, de multe ori, programul sub formă de joburi

SPMD, ca să observați dacă mai apar rezultate inconsistente (incorecte) datorate fenomenelor de *data race*.
(*Tip*: nu ar trebui să mai apară! Aceasta, bineînțeles, dacă nu veți face vreo greșeală la implementarea folosirii lacătelor, conform ideilor expuse mai sus.)

3. [MyCritSec #2bis : Parallel sorting II]
Să se modifice programul de sortare prezentat la exercițiul rezolvat **[MyCritSec #2 : Parallel sorting]** din suportul de laborator disponibil aici, astfel încât programul să primească un parametru suplimentar în linia de comandă, parametru ce va determina sensul de parcurgere a fișierului pentru efectuarea inversiunilor.
Mai exact, sensul de parcurgere va fi dat de al doilea parametru din linia de comandă cu care se va apela programul, cu următoarele valori posibile:

- -i : se vor face comparații și inversiuni numai în sensul de parcurgere `început-->sfârșit', iar când se ajunge la sfârșitul secvenței se sare la începutul ei și se reia parcurgerea de la început (adică modul implementat la exercițiul **[MyCritSec #2 : Parallel sorting]** de mai sus);
- -s : se vor face comparații și inversiuni numai în sensul de parcurgere `sfârșit-->început', iar când se ajunge la începutul secvenței se sare la sfârșitul ei și se reia parcurgerea în sens invers;
- -a : se vor face comparații și inversiuni pe ambele sensuri de deplasare, `început-->sfârșit' și `sfârșit-->început'.

Show / Hide some suggestions for solving this problem

Ideea de rezolvare: recitiți cu atenție observațiile generale și sugestiile de rezolvare date la problema rezolvată [MyCritSec #1] din suportul de laborator, disponibil aici!

4. [MyCritSec #2game : Parallel sorting III]
Să se modifice programul de sortare prezentat la exercițiul rezolvat **[MyCritSec #2 : Parallel sorting]** din suportul de laborator disponibil aici, în sensul următor: adaptați programul pentru a se comporta drept un jucător în următorul scenariu de joc:

- vor exista atâtea fișiere de sortat câți jucători participă la joc (pentru demonstrație, se admit doi jucători, adică se vor rula două instanțe ale programului);
- fiecare jucător (i.e. instanță de execuție a programului) va avea un comportament diferit ales aleator (pentru demonstrație, un jucător va trebui să ordoneze descrescător un fișier și un al doilea jucător va ordona crescător un alt fișier);
- numerele din fișiere sunt generate aleatoriu. Fiecare jucător cunoaște numele tuturor fișierelor;
- fiecare jucător pornește în ordonarea unui fișier (distinct pentru fiecare jucător);
- fiecare jucător primește câte 1 punct pentru fiecare 'nepotrivire' rezolvată, adică pentru fiecare pereche 'nepotrivită' în raport cu ordinea aleasă de acel jucător (i.e. perechi de numere aflate în ordine inversă celei alese) pe care a întâlnit-o în fișierul prelucrat și a inversat-o;
- la întâlnirea în fișierul prelucrat curent a 5 perechi 'nepotrivite' în raport cu ordinea aleasă de acel jucător, jucătorul va schimba fișierul de ordonat, adică va începe să sorteze următorul fișier (în ordine circulară a numelor de fișiere, pentru care avem relația de ordonare lexicografică pe cuvinte). Totodată, jucătorul va afișa un mesaj corespunzător, împreună cu PID-ul lui;
- fiecare jucător (i.e. instanță de execuție a programului) își va termina execuția la câștigarea a 50 de puncte sau atunci când fișierul asupra căruia lucrează este deja ordonat complet în funcție de ordinea luată de acesta în calcul. Totodată, jucătorul va afișa un mesaj corespunzător pentru fiecare situație, mesaj însoțit de PID-ul lui.

(Indicație: se va folosi primitiva fcntl() pentru punerea de lacăte pe porțiunile minimale, dar necesare pe care lucrează la un moment dat un jucător.)

Show / Hide some suggestions for solving this problem

Ideea de rezolvare: recitiți cu atenție observațiile generale și sugestiile de rezolvare date la problema rezolvată [MyCritSec #1] din suportul de laborator, disponibil aici!