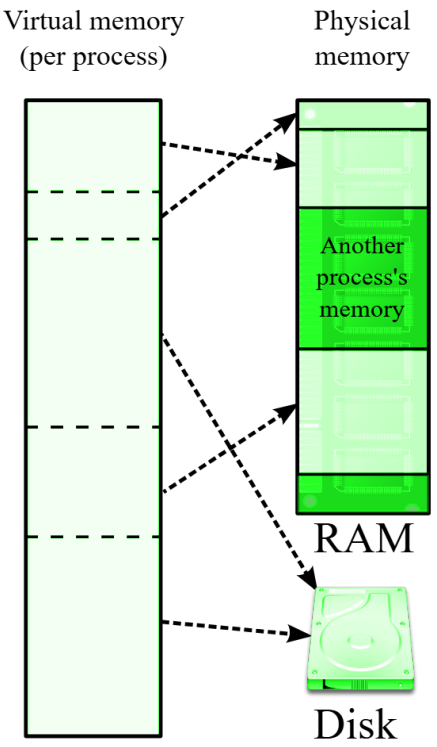
Simulator de memorie virtuala



Structura sistemelor de calcul

Universitatea Tehnica din Cluj-Napoca

Student: Moldovan Dana

Cuprins

Introducere

Context

Acest proiect are ca scop realizarea unui program de simulare grafica a operațiilor care au loc pentru gestioarea memeoriei prin memoria virtuala. Principalele operatii sunt:

- regăsirea informației
- plasarea unei pagini în memoria principala
- înlocuirea unei pagini

1.2 Obiective

Simulare isi doreste a fi una cu un scop eduactional, pentru a înțelege mecanismele de funcționare a memoriei virtuale într-un sistem de operare, prin intermediul unui simulator implementat în Java. Utilizatorul va putea simula mecanismele de gestionare a memoriei virtuale, incluzând:

- personalizarea inputului (parametrii memoriei)
- traducerea adreselor virtuale în adrese fizice.
- funcționarea TLB (Translation Lookaside Buffer).
- gestionarea tabelului de pagini.
- tratarea page fault-urilor și algoritmi de înlocuire a paginilor.
- demonstrarea algoritmului LRU (Least Recently Used) pentru înlocuirea paginilor în memoria principală și în TLB.
- dezvoltarea unei interfețe grafice intuitive care să permită vizualizarea în timp real a tuturor etapelor de acces la memorie:
- afișarea stării memoriei fizice (RAM) și a discului.
- monitorizarea ratelor de hit și miss.
- posibilitatea de a modifica o pagina care a fost adusa din memoria de pe Disk

- validarea corectitudinii simulării prin teste ample care să acopere scenarii diverse

Limbajul de programare ales pentru realizarea proiectului este Java, cu Swing pentru elemente grafice.

Studiu bibliografic

- **Managementul memoriei**
  Procesorul poate adresa direct doar regiștrii si memoria principala (RAM). Datele care nu sunt iin memorie trebuie aduse in memorie pentru a putea fi folosite, deci atat codul cat si datele programelor trebuie sa fie in memorie, codul fiind de asemenea un tip de date. Parte din Sistemul de Operare tine evidenta memoriei utilizate, precum:
  - care zone de memorie sunt ocupate si care sunt libere
  - carui proces apartine o anumita zona de memorie
  - aloca memorie pentru procese atunci cand este nevoie si o dealoca atunci cand nu mai e necesara
  - ofera protectie la accesul memoriei (cu suport hardware)
  - ofera programelor un model abstract al memoriei.

Un obiectiv foarte important al managementului memoriei este utilizarea cat mai eficienta a memoriei fizice, astfel incat mai multe procese sa poata rula simultan fara probleme. Pentru asta, se folosesc tehnici precum paginarea si memoria virtuala.
Memoria virtuala permite unui proces sa ruleze chiar daca nu toate paginile sale se afla in RAM, prin maparea adreselor virtuale la adrese fizice.

- **Memoria virtuala si spatiul de adrese**

  Exista și sisteme fara memorie abstractizata, însă în acest mod este foarte dificil sa rulam mai multe programe simultan (ar trebui de exemplu sa salvam întregul conținut al memoriei pe disc, și sa incercam memoria

următorului proces; sau sa alocam un cod pentru fiecare proces, astfel incat procesorul nu permite accesul la memoria marcată cu o alta cheie.) Din aceste motive, se foloseste memoria virtuala.

Spatiul de adrese reprezinta multimea tuturor adreselor pe care le poate folosi un proces pentru a adresa memoria. Astfel, se abstractizeaza conceptul de memorie, fiecare proces avand propriul spațiu de adrese. Adresele folosite într-un program se numesc adrese virtuale și formează spațiul de adrese virtuale. Adresele din memoria RAM formează spațiul de adrese fizice. Procesorul are o componenta numita MMU (Memory Management Unit), ce mapeaza adresele virtuale la adrese fizice.

Exista doua metode folosite pentru alocare:

- Alocarea continua: fiecarui proces i se aloca in memorie o regiune continua. Se folosesc doi registri (specifici fiecarui proces): base (adresa de inceput in memoria fizica) si limit (dimensiunea memoriei procesului). Exista anumite dezavantaje, printre care mentionez dimensiunea memoriei unui proces, care trebuie stabilita de la inceput si strategiile de alocare complexe: first-fit, next-fit, best-fit.
  Fragmentarea este:
  - externa: spatiul liber este impartit in mai multe regiuni mici, poate fi compactat
  - interna: de obicei memoria se aloca in blocuri mai mari, ramane spatiu nefolosit in interiorul unui bloc alocat

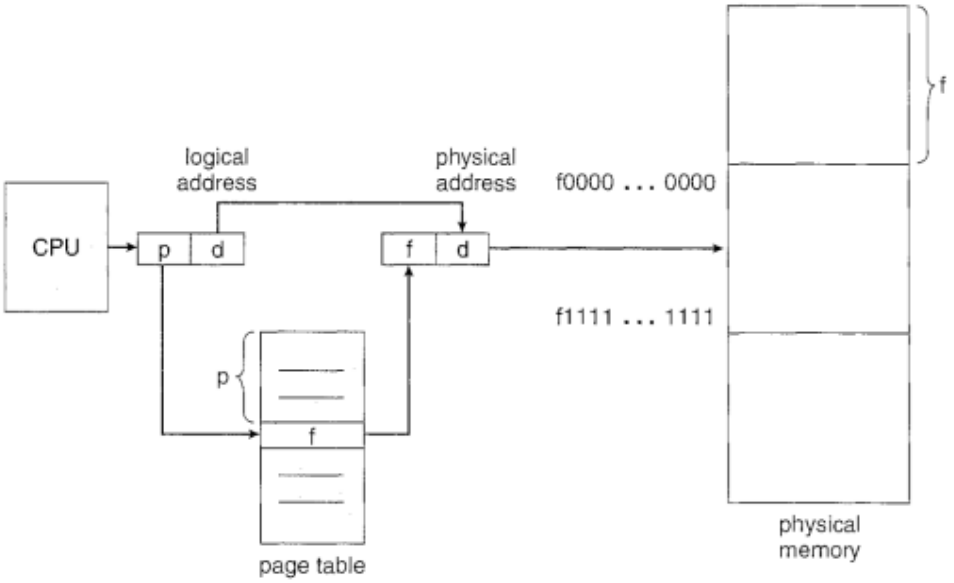
- Alocarea discontinua (paginarea)

- **Conceptul de paginare:**

Spațiul de adrese virtuale este împărțit in pagini de dimensiune fixa (4KB pe sistemele moderne), fiecare pagina contine un interval continuu de adrese.

Spațiul de adrese fizice este impartit in cadre de aceeasi dimensiune fixa cu paginile. Anumite pagini din memoria virtuala sunt mapate la cadre din memoria fizica. Se evita fragmentarea externa, orice pagină poate fi mapata la orice cadru nu mai e nevoie sa avem continuitate. Fiecare adresa virtuala este impartita in: numarul paginii (p) - bitii de ordin superior, deplasamentul in cadrul paginii (d) - bitii de

ordin inferior. Numărul paginii p este cautat in tabela de paginare si se obtine numarul cadrului corespunzător f concatenand bitii din f si d. Se obtine adresa fizica.



figură preluată din [OSC]

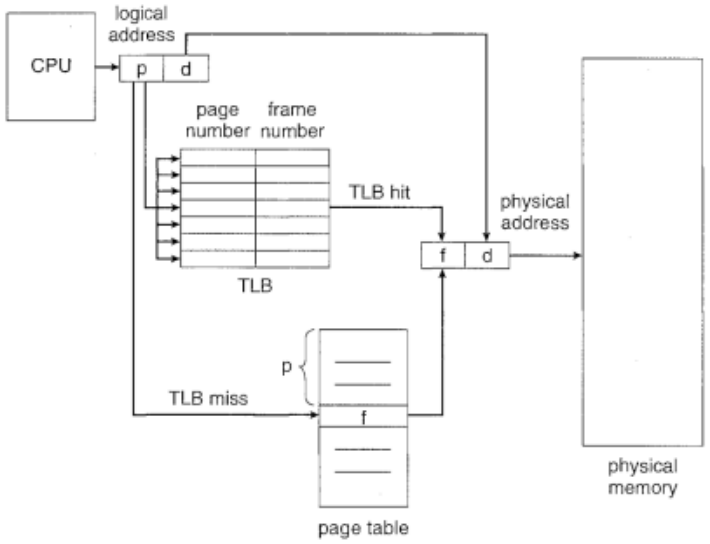
- Suport hardware pentru paginare:
  - Stocarea tabelului în regiștrii: accesul este foarte rapid, însă funcționează doar pentru tabele foarte mici, (numărul de regiștri e limitat) nu e fezabil în practică.

  - Stocarea tabelului în memorie: un registru numit PTBR (Page Table Base Register) reține adresa de început, funcționează pentru tabele mari, însă prezintă un dezavantaj: timpul de acces la memorie se dublează pentru fiecare acces la memorie trebuie sa accesam și tabelul de paginare, care e tot în memorie.

  - utilizarea TLB (Translation Look-aside Buffer): memorie rapidă, asociativă. O intrare în TLB consta din cheie (numărul paginii p) valoare (numărul cadrului corespunzător f + alte informații). Se pot reține doar o parte din pagini în aceasta memorie. Daca o pagina nu este gasita (TLB miss) se cauta in memoria principala si se aduce in TLB.

  În mod real, se folosesc tabele de paginare pe mai multe nivele: primii biti din adresa indica poziția în primul tabel intrarea din

primul tabel ne duce la adresa de început a tabelului secundar următorii biti din adresa ne indica poziția în tabelul secundar dacă nici o pagina dintr-un tabel secundar nu e folosită, acesta nu apare în memorie.



figură preluată din [OSC]

Operațiile principale în memoria virtuală

Într-un sistem cu memorie virtuală, interacțiunea dintre procesor, sistemul de operare și memoria principală presupune o serie de operații fundamentale. Cele mai importante sunt regăsirea informației, plasarea unei pagini în memoria principală și înlocuirea unei pagini.

Regăsirea informației

Atunci când un proces execută o instrucțiune care accesează o adresă virtuală, sistemul verifică dacă pagina corespunzătoare se află în memoria principală. Această verificare se face prin consultarea tabelei de pagini sau a TLB-ului (Translation Lookaside Buffer).

- Dacă pagina este prezentă în memorie, are loc un page hit, iar informația este regăsită rapid.

- Dacă pagina nu se află în memorie, se produce o eroare de pagină (page fault), iar sistemul trebuie să o aducă din memoria secundară.

Această operație reprezintă primul pas în gestionarea accesului la memorie și determină dacă este necesară o încărcare nouă în RAM.

Plasarea unei pagini în memoria principală

Dacă o pagină solicitată nu este disponibilă în RAM, sistemul de operare trebuie să o plaseze în memoria principală.

- Se caută un cadru liber în RAM.

- Pagina este copiată din memoria secundară (de exemplu, de pe disc) în acel cadru.

- Tabela de pagini și TLB-ul sunt actualizate pentru a reflecta noua corespondență între pagina virtuală și cadrul fizic.

Această etapă asigură faptul că procesul poate continua execuția fără întreruperi.

Înlocuirea unei pagini

Dacă în momentul unei erori de pagină nu există cadre libere în memoria principală, sistemul de operare trebuie să elibereze un cadru existent, adică să efectueze o înlocuire de pagină.
Alegerea paginii care va fi eliminată se face conform unui algoritm de înlocuire, cum ar fi:

- FIFO (First In, First Out) - înlocuiește pagina care se află de cel mai mult timp în memorie;

- LRU (Least Recently Used) - înlocuiește pagina care nu a mai fost utilizată recent;

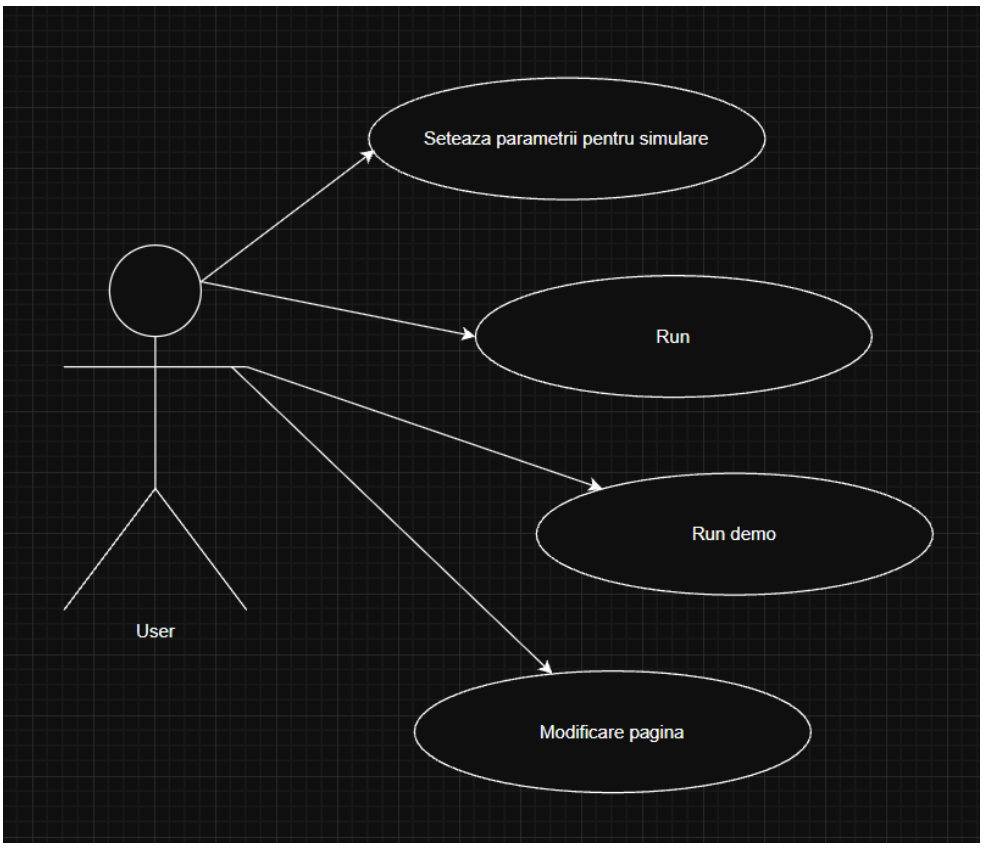
- Optimal - elimină pagina care nu va mai fi folosită cel mai mult timp în viitor.

Pagina înlocuită este salvată în memoria secundară dacă a fost modificată (dirty page), iar noua pagină este încărcată în cadrul respectiv. După actualizarea tabelei de pagini, procesul poate continua execuția de unde s-a oprit.

Analiza

Se dorește simularea memoriei virtuale cu paginare, din adrese virtuale in adrese fizice.

- **Propunere proiect**
  Simularea acceptă ca input adrese virtuale. Din acestea se extrage numărul pagini și offsetul. Se va cauta mai intai in TLB, dacă nu existam se cauta in Page Table. Apoi, daca pagina nu este in memorie, se incarca. Acum, din nou se va incerca accesarea paginii respective, de data aceasta cu succes.
  Tot acest proces va fi vizibil in maniera grafica.



- **Analiza proiect**
  Pentru a implementa o astfel de simulare, va fi nevoie de urmatoarele entitati:

  1. Pagina (Page): Aceasta va contine cheia paginii, deplasamentul, indicele pentru validitate, dirty bit si un indicator pentru timpul in care a fost folosit ultima data.

  2. Transition Look Aside Buffer (TLB): Acesta se comporta ca un cache, pentru unele corespondențe pagina - cadru. In cazul in care pagina nu este gasita aici, atunci trebuie cautata in tabela de paginare. Altfel, se pune la dispozitie cadrul care corespunde paginii care este cautata.

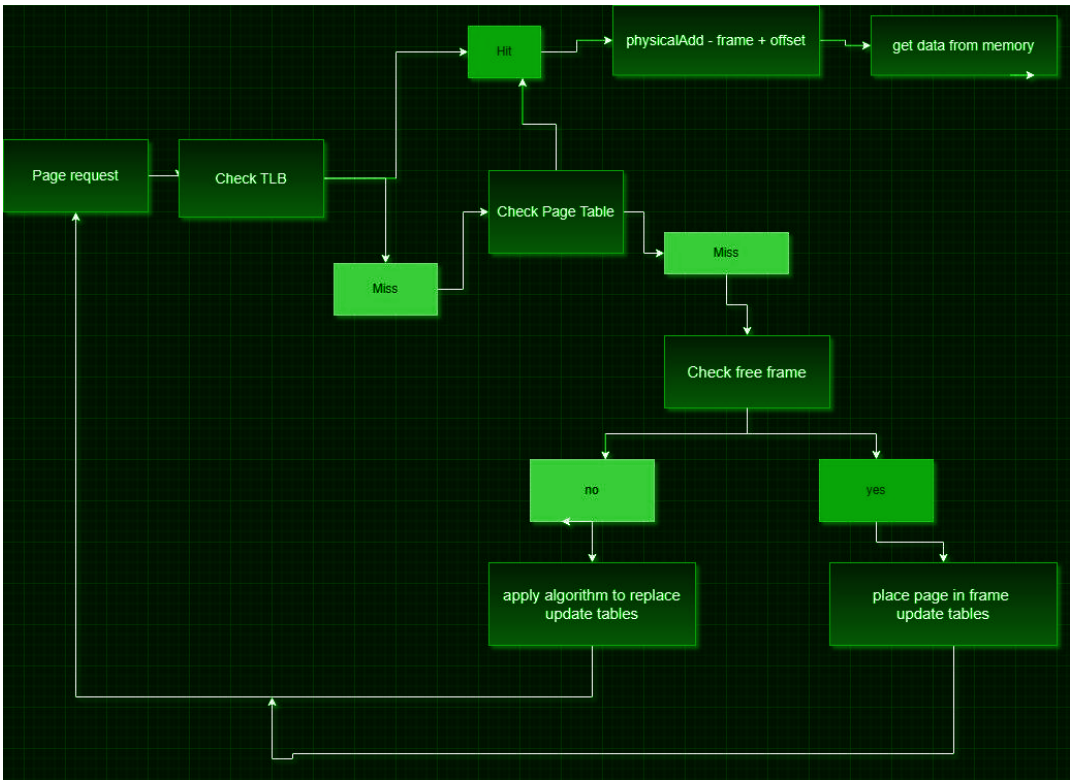
  3. Page Table: Aceasta mapeaza paginile la cadre, in acelasi fel, insa se pun la dispozitie mai multe informatii despre paginile retinute aici, cum ar fi: validitatea, dirty, lastTime Used (cand a fost folosit

ultima data, pentru a putea folosi algoritmul LRU atunci cand o pagina nu este gasita in tabela si trebuie sa realizam inlocuirea).

4. Algoritm pentru replasare: In cazul in care nu mai exista cadre libere pentru o pagina care se doreste a fi introdusa, se va cauta o pagina care poate fi scoasa din memorie, cu ajutorul agoritmului Least Recently Used (LRU), adica cadrul care corespunde paginii care a fost folosita de un timp mai mare va fi folosit pentru pagina noua

5. Memory Manager: Aceasta realizeaza fluxul care are loc mereu atunci cand se doreste gasirea informatiei din memorie.

6. Memoria fizica: Aceasta contine datele corespunzătoare pentru un anumit cadru fizic care se obtine, dupa toate operatiile.



Design

- **Arhitectura**
  Simulatorul de memorie va fi implementat in Java, fiecare entitate va corespunde unei clase.
  Principalele componente:
  - Page: contine identificatorul unic al paginii, deplasamentul in cadrul paginii, variabila pentru a arata valabilitate (va fi 1, daca pagina se afla in RAM, respectiv 0, daca e pe Disk), variabila pentru dirty, idica daca pagina a fost modificata dupa ce a fost adusa de pe disk. S-a mai folosit si variabila lastTimeUsed pentru a tine cont de momentul in care s-a accesat pagina ultima data.

  - TLB: contine o structura care mapeaza cheia paginii cu un cadru corespunzator din memoria fizica

  - Physical Memory: contine o structura care mapeaza cadrul cu un String, ce reprezinta datele care se afla la cadrul respectiv.

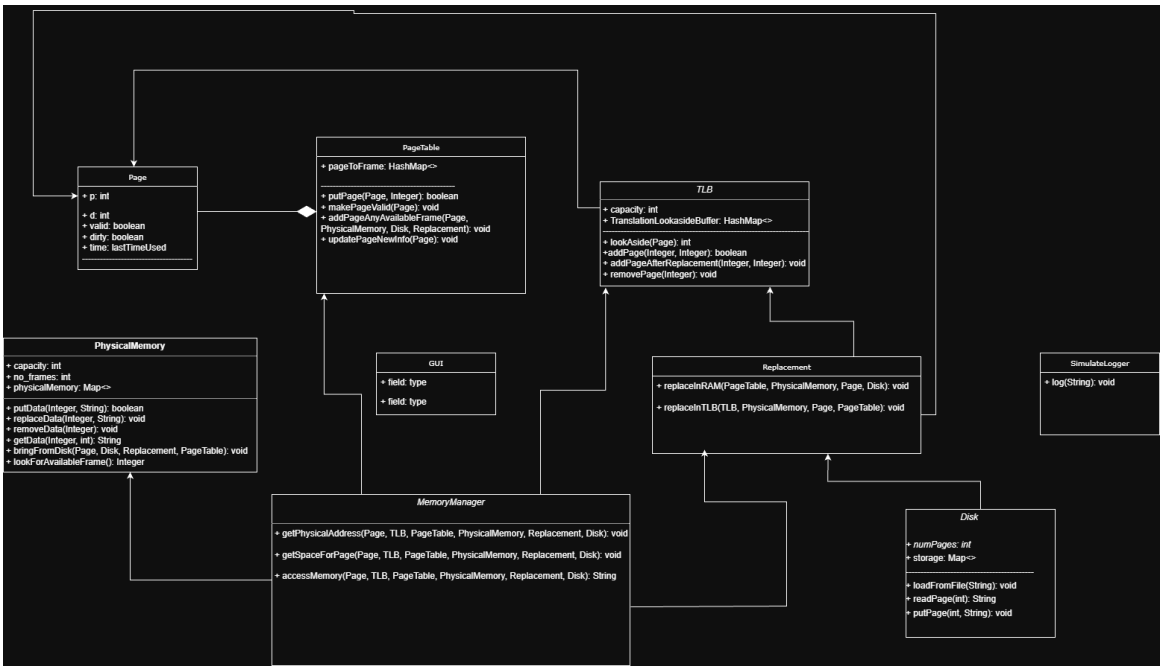
  - Disk: contine o structura care mapeaza adresa paginii la datele corespunzatoare de pe disk.

  - Memory Manager: contine metode care gestioneaza operatiile care au loc atunci cand se cere accesarea unei pagini:
    - verificare existentei in TLB sau Page Table
    - gestionarea unui page fault: aducerea de pe disk a unei pagini
    - actualizarea structurilor

Fata de aceste clase care corespund unor entități, s-au implementat și următoarele clase pentru a simula comportamentul sistemului de operare care gestionează memoria:

- Simulator: apelează metode din alte clase, pentru a simula comportamentul
- Replacement: contine algoritmul de inlocuire in caz de spațiu insuficient, pe baza politicii Least Recently Used (LRU).

- **Componente**



Pe langa clasele implementate pentru a reprezenta entitățile de baza pentru a simula o memorie virtuala, s-au realizat alte clase pentru a vizulizagrafic procesul care are loc atunci cand se gaseste o pagina (HIT), respectiv cand nu este gasita, si trebuie adusa de pe disk (MISS).

Clasa SimulationLogger va dispune mesaje pentru a anunta utilizatorul ce fel de operatie a avut loc, si locul in care a fost gasita pagina. Din fiecare clasa, se pot trimite mesaje in functie de operatia care a avut loc, si transmise mai departe pentru a fi vizualizate.

De asemenea, toate informațiile despre tabele (TLB, Page Table, Memoria Ram, paginele de pe disk) sunt transmise catre clasa care realizeaza partea de simulare grafica, astfel incat utilizatorul sa poata vedea in mod real stările entitatilor.

Implementare

- **Structurile de date folosite**
  - Tabelul de pagini (Page Table)

S-a folosit un Map<Page, Integer>, pentru a mapa paginile la un numar, care reprezinta cadrul in memorie la care corespunde

- Memoria principala (RAM)
  S-a folosit un Map<Integer, String>, pentru a mapa cadrele care corespund unor pagini, la niste date care se afla in acea corespundenta

- Memoria Secundara (Disk)
  S-a folosit un Map<Integer, String> storage, pentru maparea paginilor la datele corespondente de la acea adresa

- Pagina
  Pentru aceasta entitate, s-a folosit o structura care retine date despre o pagina:

```
private int p;
private int d;
private boolean valid;
private boolean dirty;
private Time lastTimeUsed;
```

  p- reprezinta numarul paginii, d - reprezinta deplasamentul in interiorul unei adrese, valid - da informatii in legatura cu gasirea acelei pagini in memoria principala sau nu, dirty - se foloseste pentru a lua in vedere faptul ca pagina a fost modificata de cand a fost adusa pe disk(nu este cazul pentru simularea de fata care s-a implementat), lastTimeUsed - precizeaza cand a fost folosita ultima data acea pagina, pentru a putea folosi algoritmul de inlocuire a unei pagini Late Recently Used.

- TLB
  S-a folosit un HashMap <Integer, Integer> TranslationLookasideBuffer; pentru a mapa un numar al paginii, la un cadru la care corespunde.

- **Traducerea adresei (input introdus de utilizator)**
  In tot procesul de cautare a unei pagini, se foloseste numarul paginii, deplasamentul se considera 0, in aceasta implementare. Atunci

cand utilizatorul introduce un numar, acesta este descompus, in felul urmator:

```
int pageNumber = virtualAddress / pageSize;
int offset = virtualAddress % pageSize;

addressDecompositionLabel.setText("Page #: " + pageNumber + " | Offset #: " + offset);
```

- **Fluxul pentru gasirea paginii cautate**

Clasa MemoryManager servește drept controler al sistemului, organizand succesiunea de operații necesare pentru a transforma o adresă virtuală într-o adresă fizică și pentru a gestiona erorile de pagină (miss).

1. Fluxul Principal:

Metoda getPhysicalAddress(): Această metodă implementează algoritmul standard de traducere a adresei, combinând căutarea în TLB, Page Table și gestionarea Page Fault-urilor.

A. Căutarea Rapidă (TLB Check)

TLB Hit: Se încearcă inițial căutarea directă a mapării Page ID -> Frame ID în TLB (tlb.lookAside(pageToSearch)). Dacă se găsește o adresă fizică (physicalAdd != -1), se înregistrează un TLB Hit (saveSnapshot("Page was found in TLB.")), iar execuția se încheie

B. Căutarea Secundară (Page Table Check)

RAM Hit: Dacă TLB-ul eșuează (physicalAdd == -1), se verifică Tabelul de Pagini (pageTable.isPageInMemory(pageToSearch)). Dacă pagina este în RAM (un Page Table Hit), se extrage numărul cadrului (frameFound).

Actualizarea TLB-ului: Cadrul găsit este adăugat în TLB. Dacă TLB-ul este plin, se apelează algoritmul de înlocuire LRU al clasei Replacement (rM.replaceInTLB(...)).

In aceasta implementare, offsetul se considera 0 la acces; nu este folosit.

C. Gestionarea Erorii de Pagină (Page Fault)

Page Fault: Dacă pagina nu este găsită nici în TLB, nici în Page Table (pagina este pe Disk), se declanșează un Page Fault (hit.value=false). Se apelează pageTable.addPageAnyAvailableFrame(...), care gestionează aducerea paginii de pe disc, si logica de inlocuire dacă RAM-ul este plin. După ce pagina a fost adusă în RAM, se extrage noul frameFound și se continuă cu actualizarea TLB-ului și calculul adresei fizice.

D. Actualizarea LRU

Se realizeaza indiferent de rezultat (TLB Hit, RAM Hit sau Page Fault), la finalul metodei, se actualizează timpul de utilizare.

(LRU) : pageToSearch.setLastTimeUsed(Time.valueOf(LocalTime.now())) asigură că pagina accesată devine cea mai recent utilizată.

2. Metoda accessMemory() (Punctul de Intrare)

Această metodă servește ca punct principal de intrare pentru simulator, gestionând fluxul complet de acces și tratând cazul special al lipsei totale de memorie fizică (un scenariu de eșec).

Trimitere la Traducere: Se inițiază procesul de traducere a adresei prin apelarea getPhysicalAddress().

Dacă getPhysicalAddress() returnează physicalAdd == -1, se apelează explicit getSpaceForPage() pentru alocare.

Acces Final: Odată ce physicalAdd este valid, se simulează accesul la date din Memoria Fizică (pM.getData()).

3. Logica de Suport (saveSnapshot și getSpaceForPage)

saveSnapshot(): se creează un obiect MemoryStateSnapshot care salvează starea curentă a TLB-ului, Page Table-ului și a Memoriei Fizice (pM), alături de un mesa. Sunt colectați în lista snapshots și utilizați de GUI pentru a vizualiza grafic etapele simulării pas cu pas.

getSpaceForPage(): Această metodă tratează explicit alocarea spațiului în caz de Page Fault. Verifică dacă există un cadru liber (pM.lookForAvailableFrame()). Dacă nu există, folosește pM.bringFromDisk() pentru a face loc.

- **Implementarea algoritmului LRU**

Clasa Replacement implementează politica de înlocuire Least Recently Used (LRU), atât în Memoria Fizică (RAM), cât și în TLB, atunci când apare un page fault sau un TLB miss și structura este plină.

1. Metoda replaceInRAM()

Această metodă este apelată de MemoryManager atunci când Memoria Fizică (RAM) nu mai are cadre libere și o pagină nouă (newPage) trebuie adusă de pe Disk.

A. Identificarea Paginii de inlocuit (LRU)

Inițializare: Metoda iterează prin Tabelul de Pagini (pt) pentru a găsi prima pagină validă (isValid()).

Căutarea LRU: Se parcurg toate paginile valide din Tabelul de Pagini. Pagina cu cea mai veche valoare a câmpului lastTimeUsed este selectată.

B. Procesul de inlocuire

Marcare Invalidă: Pagina este marcată ca invalidă (min.setValid(false)).

Verificare Dirty: Se verifică bitul dirty al paginii de inlocuit.

Daca este dirty, datele curente ale cadrului sunt salvate pe disk înainte de a fi suprascrise.

C. Procesul de Inlocuire (Pagina Nouă)

Citire Disk: Se citesc datele paginii necesare (newPage) de pe disk Încărcare RAM: Datele noi sunt scrise în cadrul fizic eliberat (ff) din Memoria Fizică (pm.replaceData()).

Actualizare Tabel de Pagini: Tabelul de Pagini este actualizat (pt.updateFrame()):

Noua pagină (newPage) este asociată acum cadrului eliberat (ff) si este marcată ca validă (pt.makePageValid(newPage)).

2. Metoda replaceInTLB() (Gestiunea TLB Miss-ului)

Această metodă gestionează situația în care TLB-ul este plin și o nouă mapare PageID-FrameID trebuie adăugată.

- Simularea grafica

Clasa MemorySimulatorGUI este implementată folosind Swing (biblioteca javax.swing) și servește ca interfață principală pentru utilizator. Rolul său este de a inițializa toate componentele de bază ale memoriei, de a prelua input-ul utilizatorului (adresa virtuală) și de a vizualiza starea tuturor tabelelor pas cu pas.

Panou de Status: Afișează mesaje de notificare (statusLabel) și descrie pasul curent al simulării.

Panou de Input: Permite utilizatorului să introducă adresa virtuală (pageInput). Include butoanele de control: Fetch Page (pentru a iniția accesul) și Next Step (pentru a avansa secvența de simulare), Demo (pentru input automat - din care se pot selecta doua optiuni - Random sau Secvential).

Panou de Statistici: Afișează Rata de Hit-uri (hitMissLabel) și descompunerea adresei virtuale în număr de pagină și offset (addressDecompositionLabel).

Panouri de Vizualizare: Utilizează JTextPane-uri pentru a afișa starea în timp real a structurilor de date principale: TLB, Page Table, Physical Memory (RAM) și Disk.

Instanțierea Componentelor: Constructorul MemorySimulatorGUI inițializează toate entitățile de memorie (PM, PT, TLB, Disk) pe baza configurației primite.

Se folosește un MessageDispatcher pentru a asigura comunicarea între logica de simulare și actualizarea etichetei de status (statusLabel) a interfeței.

3. Implementarea Fluxului Pas cu Pas

Simularea se bazează pe colectarea și redarea secvențială a stărilor sistemului (Snapshots).

A. Funcția fetchPage() (Inițierea Accesului)

Prelucrează adresa virtuală introdusă și o descompune în Page ID și Offset. Sr golește lista de snapshots anterioare (MemoryManager.snapshots.clear()).

Acces la Memorie: Apelează mm.accessMemory(). Această metodă execută logica de căutare (TLB -> PT -> Page Fault) și, la fiecare punct creează și stochează un MemoryStateSnapshot.

Se dezactivează fetchButton și activează nextStepButton, pregătind redarea secvențială.

B. Funcția nextStep()

Această funcție extrage următorul MemoryStateSnapshot din coada (StepByStepMessageDispatcher).updateDisplaysFromSnapshot(snap): Starea tuturor tabelelor (TLB, PT, RAM, Disk) este actualizată pe baza datelor din snapshot-ul curent. Dacă nu mai există snapshots-uri, se încheie, iar butonul Fetch Page este reactivat.

4. Vizualizarea Stării Tabelelor

Actualizarea panourilor (updatePageTableDisplayFromSnapshot, updatePMDisplayFromSnapshot, etc.) folosește obiecte StyledDocument și SimpleAttributeSet

Cadrul sau pagina care a fost accesată în pasul curent (frameFound / pageIntroduced) este evidențiat cu o culoare (galben) pentru a arăta vizual traseul de acces al memoriei. Această clasă ofera utilizatorului o metodă interactivă de a înțelege exact cum funcționează managementul memoriei virtuale și algoritmii LRU.

- **Mecanismul de Snapshot-uri**

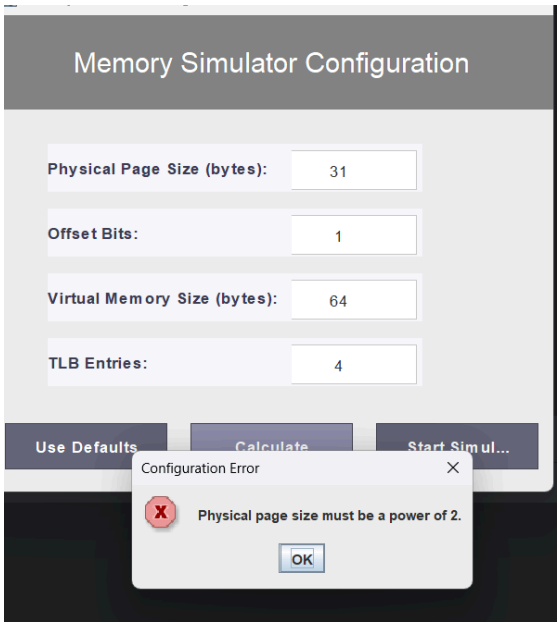
Clasa MemoryStateSnapshot: Această clasă a fost implementată pentru a captura starea imutabilă a tuturor structurilor de date (TLB, PT, PM) la un

moment dat. Fiecare obiect Snapshot conține copii ale mapărilor, permițând redarea exactă a stării sistemului la fiecare pas al simulării.

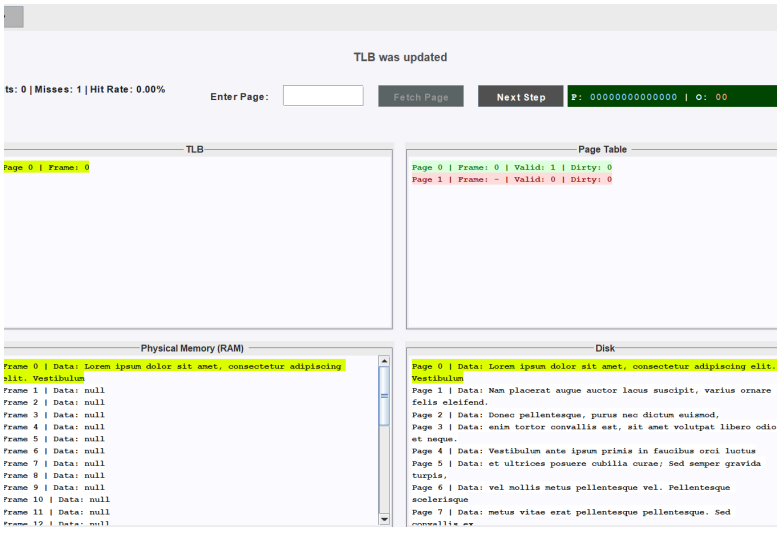
Clasa StepByStepMessageDispatcher: această clasă gestionează coada de snapshots colectată de MemoryManager, asigurând că MemorySimulatorGUI le redă în ordine cronologică la apăsarea butonului Next Step.

6. Testare și validare

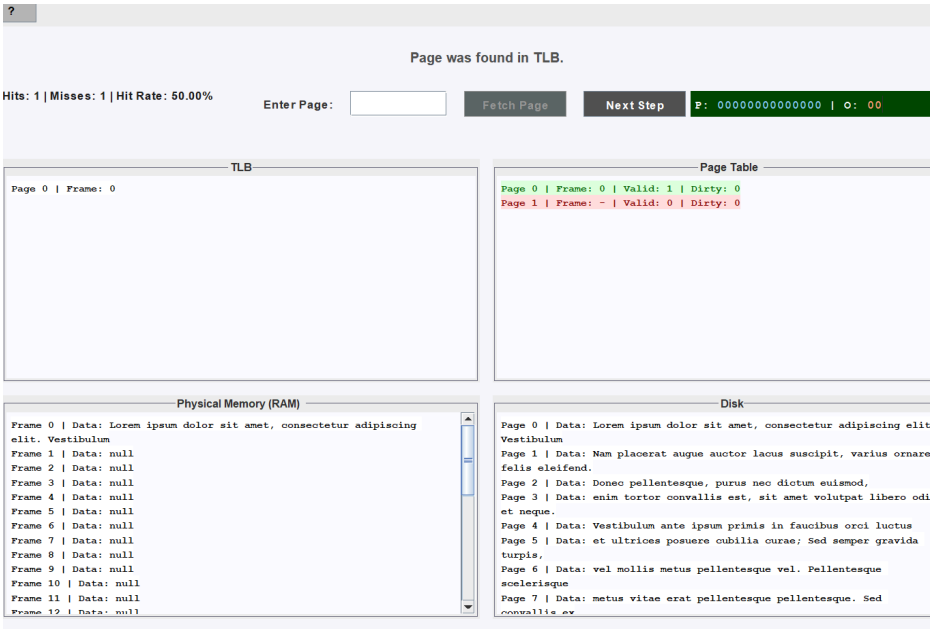
- Scenarii de test
    1) Configurarea memoriei - introducerea unor parametrii necorespunzători



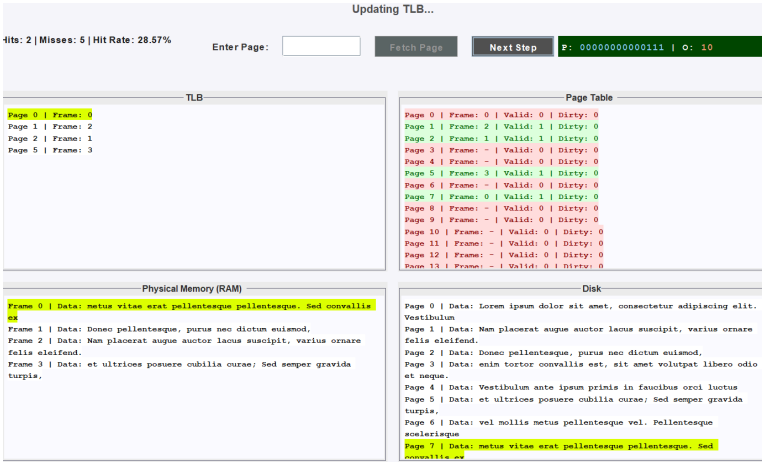
    2) Introducerea unei pagini care nu se afla in memoria RAM

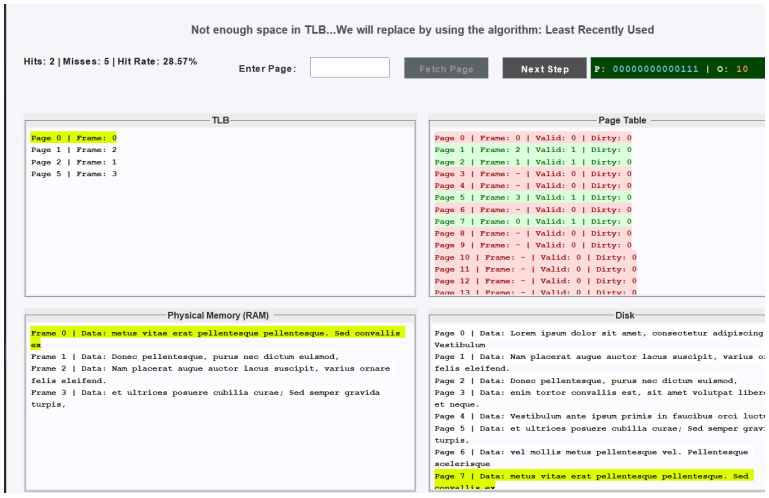


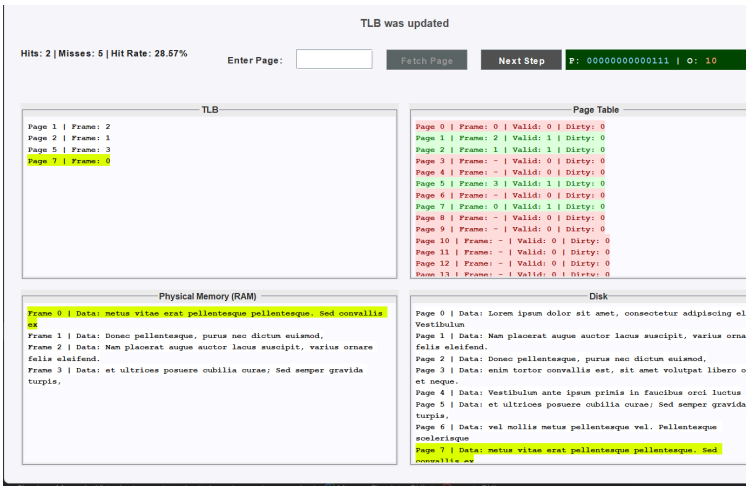
3) Introducerea unei pagini care se afla in TLB



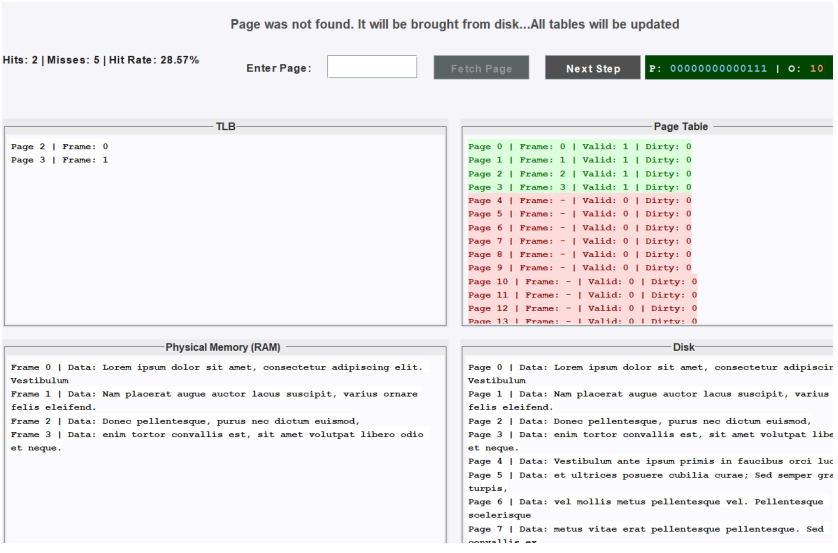
4) Introducerea unei pagini care nu se afla in TLB, dar se afla in memoria RAM + aplicarea algoritmului de inlocuire LRU in TLB

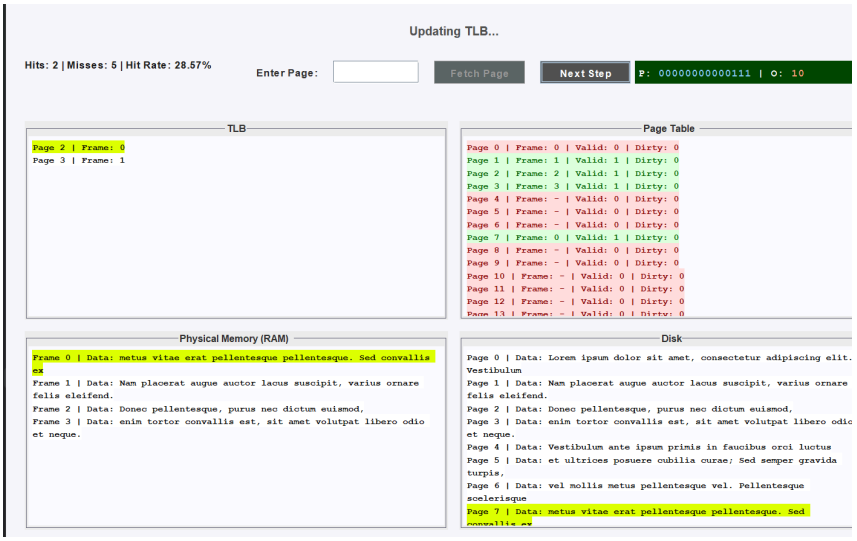






5) Aducerea unei pagini de pe disk si aplicarea algoritmului de inlocuire LRU pentru memoria RAM





Pentru o testare mai usoara automata, am adaugat optiunea de a observa modul in care se gestioneaza cautarea mai multor pagini (random sau secvential).

Concluzii

Simulatorul permite vizulizarea proceselor care au loc pentru gestiunea memoriei cu ajutorul memoriei virtuale, intr-un mod usor de inteles. Interfata grafica arata utilizatorului unde e pagina pe parcursul cautarii acesteia, aducerea in RAM, inlocuirea si aducerea inapoi pe Disk

Ca îmbunătățiri, s-ar putea adauga mai multi algoritmi pentru inlocuirea paginilor, nu doar LRU, dar si optiuni pentru o interacțiune mai buna cu simularea, precum:

- oprire
- modificarea parametrilor in timp real
- posibilitatea de a vedea istoricul accesarilor

- vizualizarea pasului anterior din procesul de gasirea a unei anumite pagini

Bibliografie

- https://personal.ntu.edu.sg/smitha/ParaCache/Paracache/vm.html
- https://dl.acm.org/doi/10.1145/3626253.3635435
- curs “Sisteme de operare”