Simulator de Memorie Virtuală (Paging)

Documentație de proiect

Păucean Ioana
grupa: 30231

Cuprins

1 Introducere

1.1 Context și domeniu

Memoria virtuală este un mecanism fundamental în sistemele de operare moderne, care permite rularea aplicațiilor într-un spațiu de adrese aparent mare și continuu, chiar dacă memoria fizică (RAM) este limitată. Implementarea clasică folosește **paginarea (paging)**: memoria virtuală este împărțită în **pagini**, iar memoria fizică în **cadre (frames)** de aceeași dimensiune. O **tabelă de pagini (page table)** păstrează mapping-ul dintre pagini și cadre.

1.2 Obiective

Obiectivele proiectului sunt:

- Simularea accesării unei pagini (regăsirea informației) și detectarea evenimentelor de tip **hit** / **page fault**.
- Simularea încărcării unei pagini în memoria principală (RAM) în cadre libere.
- Simularea înlocuirii paginilor când RAM este plină, folosind algoritmi de înlocuire:
  - FIFO (First-In, First-Out)
  - LRU (Least Recently Used)
  - OPT (Optimal – "vede în viitor")
- Realizarea unei interfețe grafice (Swing) care afișează:
  - tabela de pagini
  - cadrele din RAM
  - informații despre pasul curent (pagina accesată, hit/page fault, victimă, statistici)
- Benchmark comparativ (FIFO vs LRU, extensibil) și salvarea rezultatelor în fișiere text (`statistics.txt`, `benchmark.txt`).

2 Studiu Bibliografic

2.1 Tehnologii

- **Java** – limbaj OOP utilizat pentru implementarea logicii și a interfeței grafice.
- **Swing** – toolkit GUI standard în Java pentru construirea ferestrelor, tabelelor și componentelor interactive.
- **I/O în Java** (`FileWriter`, `PrintWriter`) – scrierea rezultatelor în fișiere text.

2.2 Algoritmi

- **FIFO**: scoate pagina care se află de cel mai mult timp în RAM (cea mai veche încărcată).
- **LRU**: scoate pagina care nu a mai fost folosită de cel mai mult timp.
- **OPT**: scoate pagina a cărei următoare utilizare este cea mai îndepărtată în viitor (sau nu mai apare deloc). Acest algoritm este folosit în simulări deoarece necesită cunoașterea secvenței de acces completă.

2.3 Biblioteci/Librării

- `javax.swing.*`: `JFrame`, `JTable`, `JButton`, `JLabel`, `JOptionPane`, `SwingWorker`
- `javax.swing.table.AbstractTableModel`: model de date pentru tabelele din UI.

3 Analiză

3.1 Arhitectură, componente, rol

Aplicația este structurată în patru pachete principale:

- **Model**: structuri de date pentru memorie (`PageTable`, `PageTableEntry`, `PhysicalMemory`, `Frame`).
- **Replacement**: interfață și implementări pentru algoritmii de înlocuire (`ReplacementStrategy`, `FifoReplacementStrategy`, `LruReplacementStrategy`, `OptimalReplacementStrategy`).
- **Controller**: motorul simulării (`MemoryManager`), rezultat per acces (`AccessResult`), și o clasă de rulare în consolă (`Simulator`).
- **GUI**: fereastra principală (`MemorySimulatorFrame`) și modele pentru tabele (`PageTableModel`, `FrameTableModel`).

3.2 Diagrama de componente (nivel pachete)

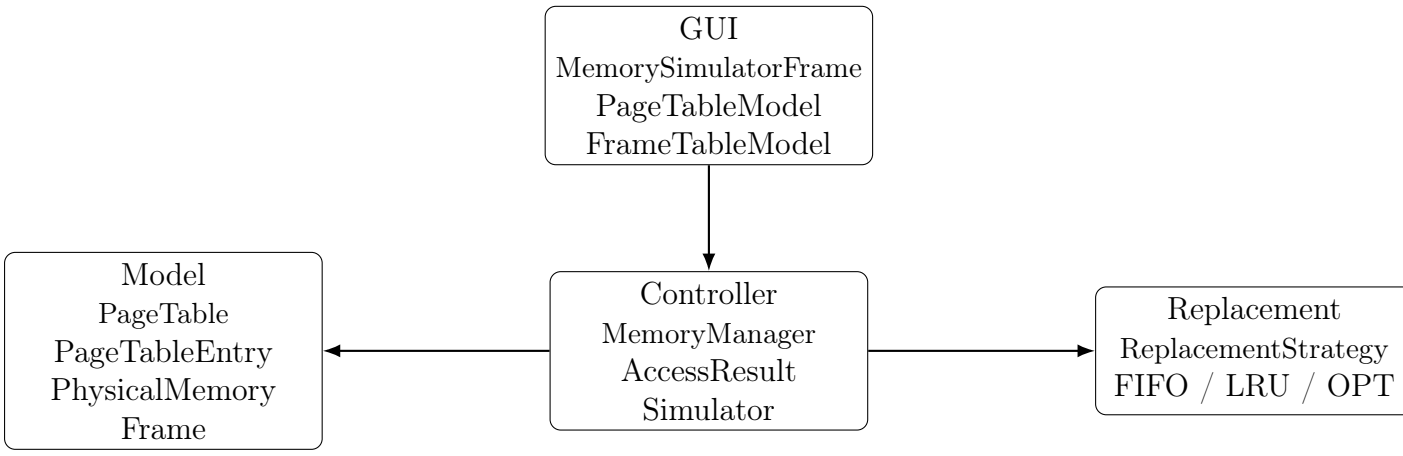

Figura 1: Arhitectura aplicației (nivel pachete).

3.3 Caracteristici funcționale

- Inițializarea simulării cu:
  - număr pagini, număr cadre
  - algoritm (FIFO/LRU/OPT)
  - secvență de acces (introducere manuală sau generare)
- Execuție **pas cu pas**: accesarea următoarei pagini din secvență.
- Execuție **Run all**: rularea întregii secvențe.
- Afișare în timp real a tabelelor: Page Table și Frames.
- Calcul statistici: total accesări, hit/miss, rate-uri, page faults.
- Salvarea statisticilor în fișiere text:
  - `statistics.txt` pentru rularea standard
  - `benchmark.txt` pentru benchmark comparativ

3.4 Caracteristici non-funcționale

- UI responsiv: benchmark-ul rulează în thread separat (`SwingWorker`).
- Reproductibilitate: benchmark cu seed fix (aceeași secvență random pentru FIFO/-LRU).
- Modularitate: algoritmii sunt implementați prin Strategy Pattern (`ReplacementStrategy`).

4 Design

4.1 Modele de date

- **PageTableEntry**: descrie o pagină virtuală (present/inRAM, frame asociat, timpi pentru FIFO/LRU).
- **Frame**: descrie un cadru fizic (index + pagina curentă).
- **PageTable**: colecție de `PageTableEntry` (indexată după numărul paginii).
- **PhysicalMemory**: colecție de `Frame` + metodă de găsire a unui frame liber.

4.2 Design Pattern: Strategy

Algoritmii de înlocuire sunt interschimbabili datorită interfeței: `ReplacementStrategy.chooseVictim(`
Controller-ul (`MemoryManager`) nu depinde de implementarea concretă, ci doar de interfață.

4.3 Fluxul de acces la o pagină (sequence)

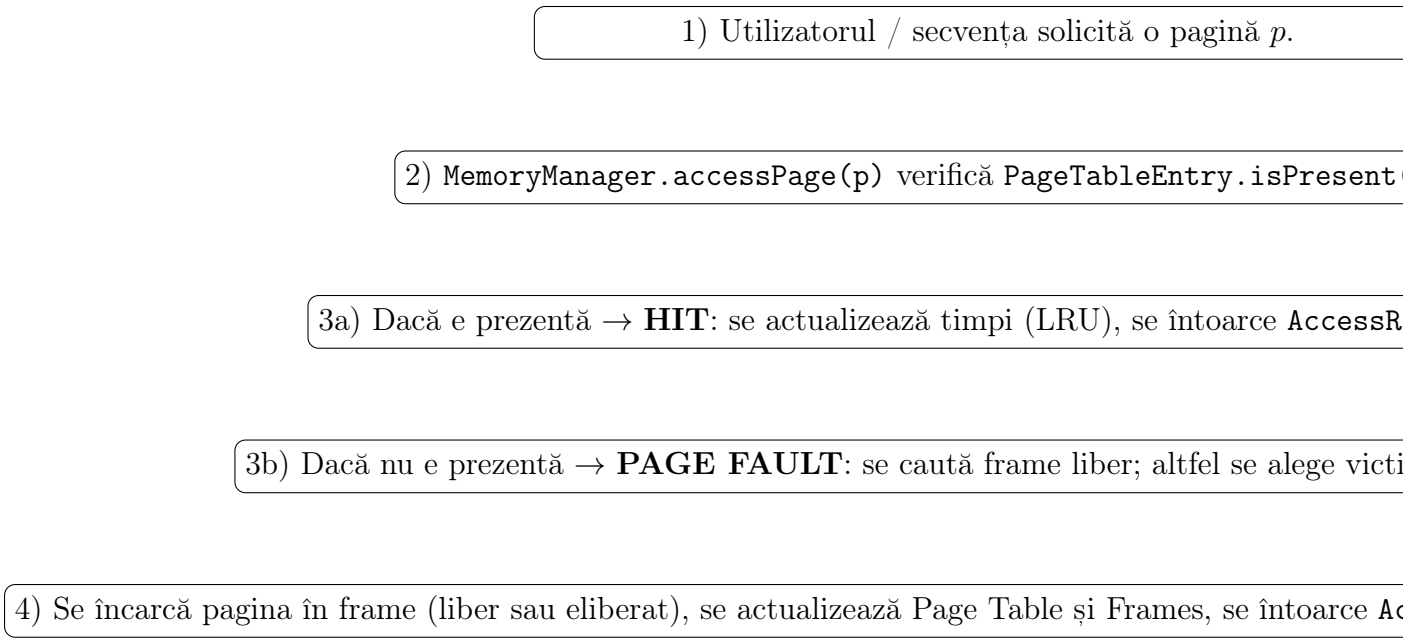

Figura 2: Fluxul logic la accesarea unei pagini.

5 Implementare

5.1 Controller: MemoryManager (fragmente relevante)

În continuare sunt prezentate fragmente de cod reprezentative din `MemoryManager`.

5.1.1 Accesarea unei pagini: hit vs page fault

```
public AccessResult accessPage(int pageNumber) {
    timeCounter++;
    PageTableEntry entry = pageTable.getEntry(pageNumber);

    if (entry.isPresent()) {
        // HIT
        entry.setLastUsedTime(timeCounter);
        return new AccessResult(pageNumber, true, null,
           pageFaultCount, timeCounter);
    } else {
        // PAGE FAULT
        pageFaultCount++;
        Integer victimPage = handlePageFault(pageNumber);
        return new AccessResult(pageNumber, false, victimPage,
           pageFaultCount, timeCounter);
    }
}
```

5.1.2 Page fault: frame liber sau înlocuire

```
private Integer handlePageFault(int pageNumber) {
    PageTableEntry entry = pageTable.getEntry(pageNumber);

    Frame freeFrame = physicalMemory.findFreeFrame();
    if (freeFrame != null) {
        loadPageIntoFrame(entry, freeFrame);
        return null; // no victim
    }

    int victimPageNumber =
       replacementStrategy.chooseVictim(pageTable);
    PageTableEntry victimEntry =
       pageTable.getEntry(victimPageNumber);
    Frame victimFrame =
       physicalMemory.getFrame(victimEntry.getFrameNumber());

    victimEntry.setPresent(false);
    victimEntry.setFrameNumber(-1);

    loadPageIntoFrame(entry, victimFrame);
    return victimPageNumber;
}
```

5.1.3 Încărcarea unei pagini într-un frame (actualizare mapping)

```
private void loadPageIntoFrame(PageTableEntry entry, Frame
   frame) {
    frame.setPageNumber(entry.getPageNumber());
    entry.setPresent(true);
    entry.setFrameNumber(frame.getFrameNumber());
    entry.setLastUsedTime(timeCounter);
    entry.setLoadedTime(timeCounter);
    entry.setDirty(false);
}
```

5.2 Replacement: FIFO, LRU, OPT

5.2.1 Interfața comună

```
public interface ReplacementStrategy {
    int chooseVictim(PageTable pageTable);
}
```

5.2.2 FIFO (alege pagina cu loadedTime minim)

```java
@Override
public int chooseVictim(PageTable pageTable) {
    int victimPage = -1;
    int oldestTime = Integer.MAX_VALUE;

    for (PageTableEntry entry : pageTable.getEntries()) {
        if (entry.isPresent() && entry.getLoadedTime() <
           oldestTime) {
            oldestTime = entry.getLoadedTime();
            victimPage = entry.getPageNumber();
        }
    }
    return victimPage;
}
```

5.2.3 LRU (alege pagina cu lastUsedTime minim)

```java
@Override
public int chooseVictim(PageTable pageTable) {
    int victimPage = -1;
    int oldestUse = Integer.MAX_VALUE;

    for (PageTableEntry entry : pageTable.getEntries()) {
        if (entry.isPresent() && entry.getLastUsedTime() <
           oldestUse) {
            oldestUse = entry.getLastUsedTime();
            victimPage = entry.getPageNumber();
        }
    }
    return victimPage;
}
```

5.2.4 OPT (vede în viitor)

OPT folosește secvența de acces completă și indexul curent pentru a alege pagina a cărei utilizare viitoare este cea mai îndepărtată (sau inexistentă).

```java
public class OptimalReplacementStrategy implements
   ReplacementStrategy {
    private int[] accessSequence;
    private int nextIndex;

    public void setAccessSequence(int[] accessSequence) {
       this.accessSequence = accessSequence; }
    public void setNextIndex(int nextIndex) { this.nextIndex =
       nextIndex; }

    @Override
    public int chooseVictim(PageTable pageTable) {
        int victimPage = -1;
```

```
11          int farthest = -1;
12
13          for (PageTableEntry e : pageTable.getEntries()) {
14              if (!e.isPresent()) continue;
15              int page = e.getPageNumber();
16              int nextUse = findNextUse(page);
17
18              if (nextUse == Integer.MAX_VALUE) return page; // never used again
19              if (nextUse > farthest) { farthest = nextUse; victimPage = page; }
20          }
21          return victimPage;
22      }
23
24      private int findNextUse(int page) {
25          for (int i = nextIndex; i < accessSequence.length; i++)
26              if (accessSequence[i] == page) return i;
27          return Integer.MAX_VALUE;
28      }
29  }
```

5.3 GUI (Swing): tabele + control execuție

Interfața folosește `JTable` cu modele custom (`AbstractTableModel`) pentru a afișa Page Table și RAM. Butoanele principale sunt: Init, Next Step, Run All, Benchmark.

5.3.1 Modele de tabel (exemplu: PageTableModel)

```
1   @Override
2   public Object getValueAt(int rowIndex, int columnIndex) {
3       List<PageTableEntry> entries =
            memoryManager.getPageTable().getEntries();
4       PageTableEntry e = entries.get(rowIndex);
5
6       switch (columnIndex) {
7           case 0: return e.getPageNumber();
8           case 1: return e.isPresent();
9           case 2: return e.getFrameNumber();
10          case 3: return e.getLastUsedTime();
11          case 4: return e.getLoadedTime();
12          case 5: return e.isDirty();
13          default: return null;
14      }
15  }
```

5.3.2 Actualizarea OPT cu indexul curent

```
if (memoryManager.getReplacementStrategy() instanceof
    OptimalReplacementStrategy opt) {
     opt.setNextIndex(currentIndex);
}
```

5.4 Benchmark (rezultate în benchmark.txt)

Benchmark-ul rulează FIFO și LRU pe aceeași secvență random (seed fix) și scrie un tabel comparativ în `benchmark.txt`.

```
Random rnd = new Random(seed);
for (int i = 0; i < accesses; i++) {
    int page = rnd.nextInt(pages);
    mm.accessPage(page);
}
```

6 Testare

6.1 Scenarii de test

1. **Hit (pagina este deja în RAM)**:
   - Config: frames suficiente, se accesează aceeași pagină repetat.
   - Așteptat: primul acces *miss*, următoarele *hit*.
2. **Page fault cu frame liber**:
   - Config: RAM are cadre libere.
   - Așteptat: pagina se încarcă într-un frame liber, fără victimă.
3. **Page fault cu înlocuire (FIFO/LRU)**:
   - Config: RAM plină (cadre ocupate).
   - Așteptat: se alege o victimă conform algoritmului; mapping-ul se actualizează corect.
4. **OPT corect pe secvență mică**:
   - Se folosește o secvență cunoscută (ex.: 0,1,2,0,3,0,4).
   - Așteptat: OPT scoate pagina utilizată cel mai târziu în viitor.
5. **Run all vs Next step**:
   - Așteptat: aceeași secvență și aceeași strategie produc aceleași statistici finale.
6. **Benchmark**:
   - Așteptat: `benchmark.txt` conține tabel FIFO/LRU și comparația (winner).

6.2 Tratarea erorilor

- Input invalid în UI (non-numeric, secvență incorectă): tratat cu `try/catch` și `JOptionPane.ERROR_MESSAGE`.
- Sfârșitul secvenței: se dezactivează butoanele de execuție și se afișează statistici.
- Erori la scriere în fișier (`IOException`): afișare mesaj de eroare și prevenirea blocării aplicației.
- Benchmark fără blocarea UI: utilizare `SwingWorker`.

7 Concluzii

Proiectul realizează o simulare completă a mecanismului de memorie virtuală bazat pe paginare: acces la pagini, încărcare în RAM, page faults și înlocuire cu algoritmi FIFO/-LRU/OPT. Arhitectura modulară (Model–Controller–Replacement–GUI) permite extinderea ușoară cu noi algoritmi (ex.: Second Chance/Clock) sau noi moduri de generare a secvențelor. Componenta de benchmark și salvarea rezultatelor în fișiere text oferă suport pentru comparații și analiză experimentală.

Bibliografie

[1] A. Silberschatz, P. Galvin, G. Gagne, *Operating System Concepts*, Wiley.

[2] A. Tanenbaum, H. Bos, *Modern Operating Systems*, Pearson.

[3] Oracle, *Java Platform Documentation* (Swing, I/O).