Dokumentation – Paging Simulator

Projektbeschreibung

Dieses Projekt realisiert eine Simulationsumgebung zur Analyse und Visualisierung von Paging-Algorithmen, wie sie in modernen Betriebssystemen zur Verwaltung des virtuellen Speichers eingesetzt werden.

Ziel des Simulators ist es, verschiedene Seitenersetzungsstrategien unter identischen, reproduzierbaren Bedingungen auszuführen und zu evaluieren. Die Bewertung erfolgt anhand aussagekräftiger Metriken wie der Anzahl der Seitenfehler (Page Faults), der TLB-Trefferquote und der mittleren Speicherzugriffszeit. Die zentrale Funktionalität umfasst die Implementierung einer Schnittstelle (API), die es erlaubt, Algorithmen wie
FIFO, Second-Chance, LRU, NRU und NFU (mit/ohne Aging) einfach zu integrieren und zu vergleichen..

Kernkomponenten

Die Simulationsumgebung bildet die wesentlichen Hardware- und Softwarekomponenten des Paging-Prozesses nach:

Physischer Speicher: Repräsentiert durch einen Vektor von Seitenrahmen (PageFrame).

Seitentabellen: Jeder simulierte Prozess verfügt über eine eigene Seitentabelle zur Übersetzung virtueller in physische Adressen.

Memory Management Unit (MMU): Koordiniert die Adressübersetzung und beinhaltet einen Translation Lookaside Buffer (TLB) mit FIFO-Strategie zur Beschleunigung von Zugriffen.

Diskrete Ereignissimulation (DES): Ein ereignisgesteuerter Kern (EventQueue) verarbeitet Speicherzugriffe aus einer Trace-Datei sequenziell, um einen realistischen und zeitlich korrekten Ablauf zu gewährleisten.

Trace-gesteuerter Ansatz

Die Simulation ist „trace-getrieben“, was bedeutet, dass die Abfolge der Speicherzugriffe aus einer externen Textdatei (trace.txt) eingelesen wird. Dies ermöglicht reproduzierbare Testläufe und eine faire Bewertung der Algorithmen. Jede Zeile in dieser Datei definiert einen einzelnen Speicherzugriff und muss dem folgenden Format entsprechen:

```
pageId [R|W]
```

- pageId: Die Nummer der virtuellen Seite, auf die zugegriffen wird.
- R oder W: Gibt an, ob es sich um einen Lese- (Read) oder Schreibzugriff (Write) handelt. Falls das Flag fehlt, wird standardmäßig ein Lesezugriff angenommen.

Jungmin Son, Oliver Jockel

Beispiel für eine trace.txt:

```
# Ein Kommentar, der ignoriert wird
0 R
1 W
2 R
0 R
```

Motivation für Lese- und Schreibzugriffe (R/W)

Die Unterscheidung zwischen Lese- und Schreibzugriffen ist für eine realitätsnahe Simulation entscheidend:

- **Realismus:** Ein Schreibzugriff markiert eine Seite als „dirty“ (modifiziert). Eine solche Seite muss beim Verdrängen zurück auf die Festplatte geschrieben werden, was höhere Kosten verursacht.
- **Algorithmik:** Algorithmen wie **NRU** (Not Recently Used) klassifizieren Seiten explizit anhand der referenced- und dirty-Bits. Ohne die Schreibinformation (W) wäre dieser Algorithmus nicht sinnvoll simulierbar.
- **Zustandsübergänge:** Ein Lesezugriff (R) setzt das referenced-Bit, während ein Schreibzugriff (W) sowohl das referenced- als auch das dirty-Bit setzt.

Kompilier- & Bedienungsanleitung

Die Anwendung besteht aus einem Kernmodul (PagingSimulator) und einer grafischen Benutzeroberfläche (PagingSimulatorUI), die den Kern als Git-Submodul einbindet.

Voraussetzungen

- **Windows:** Qt 6.x (mit Qt Creator) und CMake.
- **Linux (Debian/Ubuntu):**
  `sudo apt install build-essential cmake qt6-base-dev.`
- **macOS:**
  `brew install cmake qt.`

Kompilierung (Core & GUI)

Die empfohlene Methode ist das Klonen des UI-Projekts, da dieses den Core bereits als Submodul enthält.

1. Repository klonen (inkl. Submodul):
   ```
   git clone --recurse-submodules https://github.com/MH-Oliver/PagingSimulatorUI.git
   cd PagingSimulatorUI
   ```

2. GUI kompilieren und starten:
   a. Öffnen Sie die Datei
   b. PagingSimulatorUI.pro im Qt Creator.
   c. Wählen Sie ein kompatibles Kit aus (z.B. Qt 6.x MinGW 64-bit).
   d. Kompilieren und starten Sie die Anwendung (z.B. mit Strg+R)

Bedienung der grafischen Benutzeroberfläche (GUI)

1. **Konfiguration:** Im Hauptfenster können Parameter wie die Größe des physischen/virtuellen Speichers, die Seitengröße, die TLB-Kapazität und der gewünschte Algorithmus eingestellt werden.
   Beispielwerte: Physischer Speicher: 4096, Virtueller Speicher: 16384, Seitengröße: 1024, TLB-Größe: 2
2. **Simulation laden & ausführen:**
   a. Laden Sie eine trace.txt-Datei über den entsprechenden Button (Load simulation).
   b. Führen Sie die Simulation schrittweise (Run next page access) aus, um die Abläufe detailliert nachzuvollziehen.
3. **Visualisierung & Analyse:**
   a. Tabellenansichten: Zeigen in Echtzeit den Zustand des physischen Speichers, der Seitentabelle des aktuellen Prozesses und des TLB an.
   b. Log-Ausgabe: Ein zeitgestempeltes Log-Fenster dokumentiert jeden Schritt der Simulation, von TLB-Misses bis hin zu Seitenfehlern.
   c. Statistiken: Am Ende werden aggregierte Daten wie Zugriffe, Page-Faults, TLB-Hits/Misses und die mittlere Zugriffszeit angezeigt.

Architektur und Konzept

Dieser Abschnitt beschreibt den technischen Aufbau des Paging-Simulators, die zugrunde liegenden Designentscheidungen, die Modulstruktur und den Datenfluss innerhalb der Anwendung.

Architekturüberblick und Modulstruktur

Das Gesamtprojekt ist in zwei Hauptkomponenten unterteilt:
Die *Kernlogik* (PagingSimulator) und die *grafische Benutzeroberfläche* (PagingSimulatorUI).

Um eine saubere Trennung und Wiederverwendbarkeit zu gewährleisten, ist der PagingSimulator-Core als Git-Submodul in das PagingSimulatorUI-Projekt eingebunden. Diese Vorgehensweise stellt sicher, dass die Simulationslogik unabhängig entwickelt und potenziell auch in anderen Kontexten verwendet werden kann.

- **Kernmodule (im PagingSimulator-Submodul):**
    - core/: Enthält die zentralen Datenstrukturen (z. B. PageFrame, PageTable), die abstrakte Klasse PagingAlgorithm und alle konkreten Implementierungen der Austauschstrategien.
    - des/: Stellt die Funktionalität für die diskrete Ereignissimulation bereit, primär durch die Klassen Event und EventQueue.
    - Simulation: Die Simulation-Klasse agiert als zentraler Koordinator. Sie verwaltet den Hauptspeicher, die MMU (inkl. TLB) sowie die Prozess-Seitentabellen und sammelt Statistikdaten.
    - TraceLoader: Dieses Modul ist für das Parsen der trace.txt-Datei und das Erstellen der entsprechenden MemoryAccessEvent-Ereignisse in der EventQueue zuständig.

Simulationsansatz und Abstraktionsebenen

Das Konzept des Simulators basiert auf klar definierten Abstraktionsebenen und einem ereignisorientierten Ansatz.

- **Ereignisorientierter Ansatz (DES):** Anstelle einer kontinuierlichen Zeitachse nutzt die Simulation eine diskrete Ereignissimulation (Discrete-Event Simulation, DES). Jeder Eintrag in der Trace-Datei erzeugt ein MemoryAccess-Ereignis, das in eine EventQueue eingereiht wird. Dieser Ansatz garantiert
    - Determinismus: Bei identischem Trace und Algorithmus ist der Simulationsablauf exakt reproduzierbar.
- **Verantwortungstrennung:** Die Architektur trennt klar zwischen verschiedenen Verantwortlichkeiten:
    - Die Simulation-Klasse koordiniert den gesamten Ablauf (TLB-Lookup, Seitenfehlermanagement), pflegt die globalen Zustände (Speicher, Tabellen) und die Statistiken.
    - Das PagingAlgorithm-Interface entkoppelt die Austauschlogik vom Kern. Der Algorithmus verwaltet nur seine eigenen Metadaten (z.B. LRU-Zeitstempel oder NFU-Zähler) und muss sich nicht um die Details von Speicher- oder TLB-Updates kümmern.
    - Der TraceLoader isoliert das Einlesen und Parsen der Trace-Datei vom Rest der Simulation

API-Design und Erweiterbarkeit

Das API wurde mit dem Ziel der Klarheit und vorausschauenden Erweiterbarkeit entworfen.

- **Klares Interface (PagingAlgorithm):** Die Schnittstelle definiert einen eindeutigen Lebenszyklus mit Methoden wie memoryAccess(), selectVictimPage() und pageLoaded() . Dies macht die Implementierung neuer Algorithmen trivial.
- **Vorausschauendes Denken:** Die aktuelle Architektur ist darauf ausgelegt, zukünftige Erweiterungen mit minimalem Aufwand zu ermöglichen. Dazu gehören:
    - Komplexere Kostenmodelle: z.B. die Simulation von Schreibverzögerungen ("Write-Back-Delay").
    - Alternative TLB-Strategien: z.B. ein LRU-basierter TLB.
    - Mehrprozess-Unterstützung: z.B. die Simulation von Kontextwechseln.

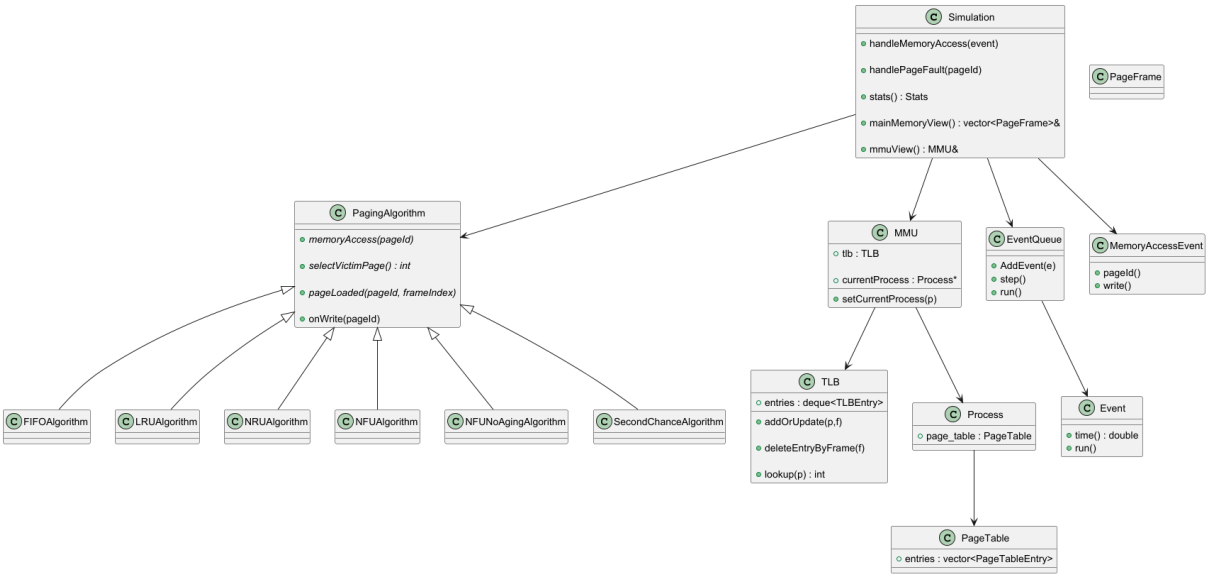
Datenfluss

Der typische Ablauf eines Simulationsdurchlaufs ist wie folgt:

1. Der TraceLoader liest einen Eintrag aus der trace.txt-Datei.
2. Ein MemoryAccessEvent wird erstellt und in die EventQueue eingereiht.
3. Die EventQueue führt das Ereignis zum geplanten Zeitpunkt aus, was die Simulation::handleMemoryAccess()-Methode aufruft.
4. Es erfolgt ein TLB-Lookup. Bei einem Fehlschlag wird die Seitentabelle geprüft.
5. Bei einem Seitenfehler (Page Fault) wird die selectVictimPage()-Methode des aktiven Paging-Algorithmus aufgerufen, um eine Seite zum Verdrängen auszuwählen.
6. Alle Aktionen (TLB-Hits, Page-Faults etc.) werden in den Statistik-Countern erfasst.

UML-Klassendiagramm

Zur Visualisierung der Beziehungen zwischen den Hauptkomponenten wurde ein UML-Klassendiagramm mittels PlantUML erstellt. Das Diagramm verdeutlicht die zentralen Abhängigkeiten:



Doxygen-Dokumentation

Der Quellcode ist durchgehend nach Doxygen-Standards kommentiert, um eine automatische Generierung der API-Dokumentation zu ermöglichen.

Generierung:
Die Dokumentation kann über die mitgelieferte Konfigurationsdatei erzeugt werden:

```
doxygen Doxyfile
```

Die Ausgabe erfolgt im HTML-Format und kann über die Datei `./docs/html/index.html` aufgerufen werden.