Übungsaufgabe

Speicherverwaltung, Virtualisierung und Speicherhierarchie: Paging, TLB, Cache, Speicherallokation

Universität:	Technische Universität Berlin
Kurs/Modul:	Systemprogrammierung
Erstellungsdatum:	September 6, 2025

Zielorientierte Lerninhalte, kostenlos!
Entdecke zugeschnittene Materialien für deine Kurse:

https://study.AllWeCanLearn.com

Systemprogrammierung

Aufgabe 1: Grundlagen der Speicherverwaltung, Virtualisierung und Speicherhierarchie

Betrachten Sie das Zusammenspiel von Paging, Translation Lookaside Buffer (TLB), Cache (L1/L2) und Speicherallokation. Die Aufgaben zielen darauf ab, Konzepte zu verknüpfen und deren Auswirkungen auf die Zugriffszeit zu analysieren.

a) Begriffsklärung und Zusammenwirken <Hier eine kompakte Übersicht zu den folgenden Begriffen liefern, ohne tiefergehende Details zu erklären:>

- Paging, Seitentabelle, Translation Lookaside Buffer (TLB)
- Cache (L1/L2) und Speicherhierarchie
- Speicherallokation und Fragmentierung

Geben Sie eine kurze Beschreibung, wie ein virtueller Zugriff schlussendlich zu einem physischen Zugriff wird und wo Caching eine Rolle spielt.

b) Paging-Grundlagen Gegeben ist ein 32-Bit-Adressenraum und eine Seitenlänge von 4 KiB. Beantworte:

- Wie viele virtuelle Seiten existieren insgesamt?
- Wie groß ist die Seitentabelle, wenn pro Page Table Entry (PTE) 4 Byte verwendet werden?
- Wie viele physische Speicherrahmen ergeben sich, wenn der physische Speicher 256 MiB umfasst?
- Nenne zwei Vorteile von Paging gegenüber einer rein segmentierten Speicherverwaltung.

c) TLB-Szenario Ein TLB mit 4 Einträgen verwendet den LRU-Ersatz. Die TLB ist zu Beginn leer. Zugriffsequenz (virtuelle Adressen, Hex) lautet:

`0x00403000`, `0x00403004`, `0x00A0B000`, `0x00A0B004`, `0x00403000`.

Bestimme die Anzahl der TLB-Hits und TLB-Misses in der gegebenen Reihenfolge.

d) Cache-Szenario Angenommen ein L1-Datencache mit

- Größe: 32 KB,
- Blockgröße: 64 Byte,
- 4-Way-Set-Associative,
- LRU-Ersatz.

Gegeben sei die Zugriffsliste auf Adressen (hex):

`0x00000000`, `0x00000040`, `0x00000080`, `0x00000000`.

Bestimme für jeden Zugriff, ob es ein Cache-Hit oder -Miss ist, und gib ggf. an, welche Blöcke im Cache liegen bzw. ersetzt würden.

e) Speicherallokation Vergleiche drei gängige Allokationsstrategien:

- First-Fit,
- Best-Fit,
- Buddy-Allocator. Für jeden Ansatz leite kurz her, wie er mit einem typischen Fragmentierungsverhalten umgeht und welche Vor- bzw. Nachteile sich ergeben. Ergänze eine kurze praxisnahe Einschätzung, wann welche Strategie sinnvoll ist.

Aufgabe 2: Übungsaufgabe zur Modellierung von Paging- und Cache-Verhalten

Diese Aufgabe dient der Veranschaulichung von Konzepten aus Aufgabe 1 und regt an, notwendige Berechnungen systematisch durchzuführen.

a) Beschreiben Sie in Pseudocode die Ablaufschritte eines Speicherzugriffs unter Einbezug von TLB-Check, Seiten-Tabellenzugriff und möglichem Page-Fault. Gehen Sie dabei spezifisch auf die Rollen von TLB, Seitentabelle und Evakuierungs-/Nachlademanagement ein.

b) Diskutieren Sie, wie sich unterschiedliche TLB-Größen und unterschiedliche Seitenlängen auf die durchschnittliche Zugriffslatenz auswirken können. Nutzen Sie dazu das Grundprinzip der TLB-Hits/Misses und der Notwendigkeit eines Page-Tabellen-Zugriffs im Falle eines TLB-Misses.

c) Skizzieren Sie zwei Varianten zur Layout-Planung eines L1-Daten-Caches (z. B. unterschiedliches Set-Associativity-Design). Welche Auswirkungen haben höhere Associativität und größere Cache-Größe analog zu den Optionen aus Aufgabe 1d?

Lösungen

Aufgabe 1: Grundlagen der Speicherverwaltung, Virtualisierung und Speicherhierarchie

a) Lösung

- Paging, Seitentabelle, Translation Lookaside Buffer (TLB): Paging teilt den virtuellen Adressraum in feste Seiten (z. B. 4 KiB) auf. Die Zuordnung von virtueller Seite zu physischem Rahmen erfolgt in der Seitentabelle. Der TLB ist ein schneller Cache, der häufig angefragte VPN→PFN-Einträge speichert, um eine vollständige Seitentabelle bei jedem Zugriff zu vermeiden.
- Cache (L1/L2) und Speicherhierarchie: Die CPU greift zunächst auf sehr schnellen Cache zu, der Kopien von Speicherblöcken enthält. Nachfolgende Ebenen (L1/L2/L3, Hauptspeicher) bilden die hierarchische Speicherorganisation. Caches arbeiten mit Adressbildung auf Blockebene und tragen so zur Reduktion der durchschnittlichen Zugriffszeit bei.
- Speicherallokation und Fragmentierung: Speicherallokation verwaltet freie bzw. belegte Blöcke. Fragmentierung kann extern (zwischen belegten Blöcken) auftreten; Paging reduziert externe Fragmentierung durch feste, gleichgroße Seiten, während Segmentierung tendenziell zu externer Fragmentierung neigt.

Geben Sie eine kurze Beschreibung, wie ein virtueller Zugriff schlussendlich zu einem physischen Zugriff wird und wo Caching eine Rolle spielt.

b) Lösung

$$\text{Virtuelle Seiten} = 2^{32-12} = 2^{20} = 1{,}048{,}576$$

$$\text{Größe der Seitentabelle} = 4\ \text{Byte} \times 2^{20} = 4{,}194{,}304\ \text{Byte} = 4\ \text{MiB}$$

$$\text{Physischer Speicher} = 256\ \text{MiB} \quad \Rightarrow \quad \text{Anzahl Physischer Rahmen} = \frac{256\ \text{MiB}}{4\ \text{KiB}} = \frac{256 \cdot 2^{20}}{2^{12}} = 65{,}536$$

Zwei Vorteile von Paging gegenüber rein segmentierter Speicherverwaltung:

Keine externe Fragmentierung, da alle Seiten die gleiche Größe haben.

Ermöglicht Demand Paging und effiziente Speichermanagement-Strategien (z. B. virtueller Speicher, Schutz, Swapping).

c) Lösung Der gegebene TLB-Plan mit 4 Einträgen (LRU) und die Sequenz lautet: 0x00403000, 0x00403004, 0x00A0B000, 0x00A0B004, 0x00403000.

VPNs:

- 0x00403000 → VPN = 0x403
- 0x00403004 → VPN = 0x403
- 0x00A0B000 → VPN = 0xA0B
- 0x00A0B004 → VPN = 0xA0B
- 0x00403000 → VPN = 0x403

Zugriffsfolgen:

- 1) Miss (TLB leer → 0x403 eingefügt)
- 2) Hit (0x403 bereits im TLB)
- 3) Miss (0xA0B neu → 0xA0B eingefügt)
- 4) Hit (0xA0B bereits im TLB)
- 5) Hit (0x403 weiterhin im TLB)

TLB-Hits: 3, TLB-Misses: 2. Die TLB-Belegung bleibt durch LRUs Ersetzung stabil, da nur zwei eindeutige VPNs verwendet werden.

d) Lösung Gegebene Architektur: L1-Datencache, 32 KiB, Blockgröße 64 Byte, 4-Way-Set-Associative, LRU.

Berechnungen:

- Block-Offset-Bits: log2(64) = 6
- Sets: 32 KiB / (64 B) = 512 Blöcke; 512 Blöcke / 4 Ways = 128 Sets
- Set-Index-Bits: log2(128) = 7
- Tag-Bits: 32 - 6 - 7 = 19

Zugriffe (hex):

$$0x00000000 \rightarrow \text{Block 0, Set 0} \quad \text{Miss}$$
$$0x00000040 \rightarrow \text{Block 1, Set 1} \quad \text{Miss}$$
$$0x00000080 \rightarrow \text{Block 2, Set 2} \quad \text{Miss}$$
$$0x00000000 \rightarrow \text{Block 0, Set 0} \quad \text{Hit}$$

Zusammenfassung:

- Cache-Hits/Misses-Reihenfolge: Miss, Miss, Miss, Hit.
- Nach dem dritten Zugriff befinden sich Blöcke 0 in Set 0, 1 in Set 1, 2 in Set 2.

e) Lösung Vergleich der Allokationsstrategien:

- First-Fit
  - Funktionsweise: Nimmt sofort den ersten freien Block, der groß genug ist.
  - Fragmentierung: Tendenziell zu externer Fragmentierung, da freie Blöcke links liegenbleiben.
  - Vorteile/Nachteile: Schnell, einfach; kann aber zu vielen kleinen Restblöcken führen.
- Best-Fit
  - Funktionsweise: Wählt den kleinsten frei verfügbaren Block, der passt.
  - Fragmentierung: Reduziert externe Fragmentierung im Durchschnitt, kann jedoch viele feine Lücken erzeugen (Fragmentierungsüberschuss durch viele kleine Blöcke).

  - Vorteile/Nachteile: Geringer Abfall pro Allocation, aber langsamer Suche, häufige Allokations-/Freigabe-Overheads.

- Buddy-Allocator
  - Funktionsweise: Speicher in Blöcken der Größe Potenzen von zwei; Blöcke werden bei Bedarf geteilt und beim Freigeben wieder zusammengeführt.
  - Fragmentierung: Sehr geringe externe Fragmentierung; Koaleszenz ist effizient.
  - Vorteile/Nachteile: Vorhersehbare Allocate-/Free-Zeiten, einfache Koaleszenz; Nachteil: interne Fragmentierung durch Rounded-Up-Größen.

Praxisnahe Einschätzung:

- First-Fit eignet sich gut für generische Heap-Verwaltung in vielen Anwendungen, wo Fragmentierung tolerierbar ist und Schnelligkeit wichtig ist.
- Best-Fit kann sinnvoll sein, wenn der Allocator typischerweise mit vielen unterschiedlichen Blockgrößen arbeitet und man möglichst wenig Restfragmentierung verursachen möchte, doch die Suche kann teuer werden.
- Buddy-Allocator ist besonders geeignet für kernelnahe Speicherpools, Page-Frames oder schnelle Allokationen mit vorhersehbaren Laufzeiten; er reduziert externe Fragmentierung effektiv, führt aber zu absorbierender interner Fragmentierung durch die festen Blockgrößen.

Aufgabe 2: Übungsaufgabe zur Modellierung von Paging- und Cache-Verhalten

a) Lösung

Ziel: Beschreiben Sie in Pseudocode die Ablaufschritte eines Speicherzugriffs unter Einbezug von TLB-Check, Seiten-Tabellenzugriff und möglichem Page-Fault.

- TLB-Check: Prüfe, ob der VPN (virtuelle Seitenummer) im TLB vorhanden ist. Wenn ja, verwende den zugehörigen PFN (physical frame number) und fahre fort; ansonsten weiter zum Seitenwa:lk.
- Seiten-Tabellenzugriff: Lese die Seitentabelle, um den PFN zu erhalten. Falls der Eintrag ungültig ist (Page Fault), rufe den Page-Fault-Handler auf (z. B. Lade die Seite aus dem Sekundärspeicher in den Hauptspeicher), aktualisiere danach die Seitentabelle.
- Evakuierungs-/Nachlademanagement: Falls der TLB voll ist, evictiere den ältesten Eintrag gemäß LRU; nach dem Page-Fault ggf. Lese-/Schreibe-Backlog beachten; Aktualisiere TLB mit dem neuen VPN→PFN-Eintrag.
- Adressbildung: Physische Adresse = PFN « 12 | Offset (Offset = lower 12 Bits der virtuellen Adresse).

```
function access(virtual_addr):
    vpn = virtual_addr >> 12
    offset = virtual_addr & 0xFFF
    if TLB.contains(vpn):
        pf = TLB[vpn]
    else:
        pf = page_table[vpn]
        if not pf.valid:
            handle_page_fault(vpn)
            pf = updated_pf_after_fault(vpn)
        TLB.insert(vpn, pf, evict_LRU=True)
    physical_addr = (pf << 12) | offset
    return read_memory(physical_addr)
```

b) Lösung Einfluss von TLBGroße und Seitengröße auf die durchschnittliche Zugriffslatenz:

- Größere TLB: geringere TLB-Misses, damit weniger Page-Table-Zugriffe (Speicherzugriffe auf das Paging-Objekt) erforderlich, was die Latenz senkt.
- Kleinere Seitenlänge (mehr Seiten): erhöht die Anzahl der Einträge in der Seitentabelle; größerer Page-Tab-Zugriffsoverhead bei TLB-Misses, aber potenziell geringere interne Fragmentierung.

- Größere Seitengrößen: weniger Seiten, damit weniger TLB-Einträge nötig; Potenzial für höhere interner Fragmentierung innerhalb einer Seite, aber geringere TLB-Missrate; Page-Table-Hops können tiefer werden, wenn die Seitentabellensstruktur unoptimiert ist. Kurz gesagt: Eine größere TLB verringert Missquoten (und damit Page-Table-Zugriffe) und reduziert die durchschnittliche Zugriffslatenz, während Seitenlänge und Seitengröße das Verhältnis von TLB-Missrate zu Seitentabellzugriff beeinflussen.

c) Lösung Zwei Varianten zur Layout-Planung eines L1-Daten-Caches (unterschiedliche Set-Associativity):

- Variante 1 (Zweiteilung wie in Aufgabe 1d): 4-Way Set-Associative, 32 KiB, Blockgröße 64 B, 128 Sets.
  - Auswirkungen: Höhere Associativität reduziert Konflikt-Misses, verbessert Trefferraten bei konkurrierenden Adressmustern; Komplexität der Adressverifikation erhöht sich leicht.
- Variante 2: 8-Way Set-Associative oder alternativ größere Cache-Größe (z. B. 64 KiB) mit 8-Way SA.
  - Auswirkungen: Noch geringere Konflikt-Misses, aber größere benötigte Hardware für Tagnachverfolgung, etwas höhere Latenz pro Zugriff aufgrund größerer Tag-Vergleiche; größere Kapazität reduziert Missrate weiter, benötigt aber mehr Chipfläche.

Schlussfolgerung: Höhere Associativität verringert Konflikt-Misses, größere Cache-Größe senkt Missrate weiter; beides verbessert die durchschnittliche Zugriffslatenz, erhöht aber Hardwarekomplexität und ggf. Zugriffslatenz pro Zugriff. Abhängig von Budget und Anwendungsprofilen wählt man eine passende Balance zwischen Größe, Geschwindigkeit und Komplexität.