

Speicherhierarchie

Dr.-Ing. Volkmar Sieh

Department Informatik 4
Systemsoftware
Friedrich-Alexander-Universität Erlangen-Nürnberg

WS 2024/2025



Beispiel:

```
addl $1, %es:4(%edx)
```

Befehl muss 7(!) Mal auf Speicher zugreifen:

- Befehlsprefix lesen
- Befehlsbyte lesen
- ModR/M-Byte lesen
- Offset lesen
- Konstante lesen
- Operand lesen
- Ergebnis schreiben



Schritte beim Speicherzugriff x86:

CPU-Core:

- Berechnung der virtuellen Adresse (hier: `%eip` bzw. $4 + \%edx$)

...



...

Segmentierungseinheit:

- Überprüfung, ob das Segment ein NULL-Segment ist.
Wenn ja => Protection fault.
- Überprüfung, ob in das Segment geschrieben werden darf.
Wenn nein => Protection fault.
- Überprüfung, ob Adresse plus Größe des Operanden (hier: 4 Byte) innerhalb (normales Segment) bzw. außerhalb (expand-down-Segment) des %es-Segmentes liegt.
Wenn nicht => Protection Fault.
- Berechnung der linearen Adresse (virtuelle Adresse plus Basis-Adresse des Segments).

...



...

Alignment-Check-Einheit:

- Überprüfung, ob der Operand korrekt aligned liegt.
Wenn nicht und Alignment-Checking aktiv ist => Alignment Fault.

...



...

Debug-Einheit:

- Überprüfung, ob
 - die Adresse mit einer der 4 Watchpoint-Adressen,
 - die Größe des Operanden mit der entsprechenden Watchpoint-Größe
 - und die Schreib-/Lese-Richtung mit den entsprechenden R/W-Bits der Debug-Einheit übereinstimmt.

Wenn ja und der Watchpoint aktiv ist => Debug Trap.

...



...

Level-1-Cache:

- Wenn Zeile nicht im Cache und keine Zeile frei, eine andere Zeile in den Level-2-Cache (mit Cache-Info) schreiben und Zeile als „frei“ markieren..
- Wenn Zeile nicht im Cache, MMU nach physikalischer Adresse und Caching-Strategie fragen:
MMU/MTRR/PAT:
 - ...
Zeilen-Info in Cache eintragen.
- Wenn Zeile „cachebar“ und nicht im Cache, Zeile vom Level-2-Cache lesen (mit Cache-Info).
- Wenn Zeile „cachebar“, entsprechendes Datum überschreiben.
Ansonsten Datum in den Level-2-Cache schreiben (mit Cache-Info).

...



■ ...

MMU/MTRR/PAT:

- Wenn virtuelle Adresse nicht im TLB, dann Level-2-Cache nach Page-Table-Einträgen (gemäß eingestelltem Modus; PAE, PSE, PSE36, LM, ...) und MTRR/PAT nach Cache-Modus fragen und im TLB eintragen.
- Überprüfung Eintrag, ob Seite für CPU im aktuellen Modus (Ring 0-3) lesbar bzw. schreibbar ist.
Wenn nein => Page fault.
- ggf. Setzen der „Accessed“- und „Dirty“-Bits.

■ ...



...

Level-2-Cache:

- Wenn Datum „cachebar“ und Zeile nicht im Cache und keine Zeile frei, eine andere Zeile auf den Bus rausschreiben (Burst-Zugriff) und Zeile als „frei“ markieren.
- Wenn Zeile „cachebar“ und nicht im Cache, Zeile vom Bus lesen (Burst-Zugriff).
- Wenn Zeile „cachebar“, entsprechendes Datum überschreiben. Ansonsten Datum über den Bus rausschreiben (Einzel-Transaktion).

Bus:

...

Komponente:

...



Zusammenfassung:

Speicherzugriffe durchlaufen viele Einheiten, die

- die Parameter überprüfen,
- die Parameter in andere umrechnen.

Will man das komplett korrekt nachmachen, wird die Performance der Emulation **katastrophal** sein!

Oder...?



Idee: Caches weglassen:

Sie kosten bei der Emulation Zeit; bringen keinen Vorteil.

Caches sind sowohl für das Betriebssystem als auch für die Anwendungen weitgehend unsichtbar.

Aber:

- Sie müssen z.B. vom Betriebssystem richtig eingestellt werden.
- Sie beeinflussen maßgeblich die Performance des Systems.
- In Multiprozessor-Systemen oder Systemen mit DMA werden sie u.U. sichtbar.



Idee: JIT-Compiler:

- Segmentierungs-Checks nur dazu-compilieren, wenn Segment-Limit (z.Z.) wirklich limitiert (\neq 4GByte ist)
- Addition der Segment-Basis-Adresse nur dazu-compilieren, wenn (z.Z.) Basis-Adresse \neq 0
- Alignment-Checking nur dazu-compilieren, wenn (z.Z.) Alignment-Checking angeschaltet ist
- Watchpoint-Checks nur dazu-compilieren, wenn (z.Z.) Watchpoints aktiv sind
- ...

Code entsprechend invalidieren, wenn sich eine der Bedingungen ändert.



Hilft im Standard-Fall (Linux/Windows) sehr (keine Segmentierung, Alignment-Checking meist aus, meist kein Debugging aktiv).

Hilft z.B. beim Emulieren von MSDOS/FreeDOS (fast) gar nicht (Segmentierung wird intensiv genutzt).



Idee: MMU:

- MMU macht Lookup in TLB
- Memory-Info-Caching macht Lookup im Info-Cache

Lookups verknüpfen:

- MMU macht Lookup in TLB
- im TLB stehen nicht mehr physikalische Adressen sondern direkt die Pointer zu den emulierten Speicherseiten

=> Speicherzugriff über einen, schnellen Lookup!



Die letzte Idee läßt sich verallgemeinern:

Genutzt wird eine Art Cache,

- der über die Parameter des Speicherzugriffs (Read/Write/Execute, Privilegstufe, Segment-Nummer, Segment-Offset) indiziert wird,
- der direkt den Zeiger auf dem emulierten Speicher enthält.

Zeiger werden nur dann eingetragen, wenn alle Checks den (direkten) Zugriff erlauben.



Speicherhierarchie – Idee

```
static uint8_t *mem[4][6][1 << 32];

void
store(int ring, int seg, uint32_t offset, uint8_t val) {
    uint8_t *ptr;

    ptr = mem[ring][seg][offset];
    if (ptr == NULL) {
        mem[ring][seg][offset] = get_ptr(ring, seg, offset);
        ptr = mem[ring][seg][offset];
    }
    if (ptr == NULL) {
        store_slow(ring, seg, offset, val);
    } else {
        *ptr = val;
    }
}
```



Array mem i.A. zu groß (im Beispiel: 768 GByte)...

Daher ähnliche Struktur zum Cachen. Z.B.:

```
#define COUNT 4096

static struct entry {
    uint32_t offset;
    uint8_t *ptr;
} mem[4][6][COUNT];
```



Speicherhierarchie – Idee

```
void
store(int ring, int seg, uint32_t offset, uint8_t val) {
    int hash = offset % COUNT;
    struct entry *ent = &mem[ring][seg][hash];
    uint8_t *ptr;

    if (ent->offset != offset) {
        ent->offset = offset;
        ent->ptr = get_ptr(ring, seg, offset);
    }
    ptr = ent->ptr;
    if (ptr == NULL) {
        store_slow(ring, seg, offset, val);
    } else {
        *ptr = val;
    }
}
```



Meistens gilt:

```
get_ptr(ring, seg, offset + 1)  
    == get_ptr(ring, seg, offset) + 1;
```

(Ausnahme: Seitengrenzen)



Speicherhierarchie – Idee

```
#define MASK 0x1f

void
store(int ring, int seg, uint32_t offset, uint8_t val) {
    int hash = (offset / (MASK + 1)) % COUNT;
    struct entry *ent = &mem[ring][seg][hash];
    uint8_t *ptr;

    if (ent->offset != offset & ~MASK) {
        ent->offset = offset & ~MASK;
        ent->ptr = get_ptr(ring, seg, offset & ~MASK);
    }
    ptr = ent->ptr;
    if (ptr == NULL) {
        store_slow(ring, seg, offset, val);
    } else {
        *(ptr + (offset & MASK)) = val;
    }
}
```



Alignment-Check auch noch integrierbar.

Z.B. Check für 4-Byte-Werte:

```
...  
if (ent->offset != offset & (~MASK | 0x3)) {  
    ...  
}  
...
```



Callback-Funktionen auch noch integrierbar:

```
#define COUNT 4096

static struct entry {
    uint32_t offset;
    uint32_t paddr;
    uint8_t *ptr;
    void (*cb)(void *s, uint32_t paddr, uint8_t val);
    void *s;
} mem[4][6][COUNT];
```



```
...
if (ent->offset != offset & (~MASK | 0x3)) {
    ...
    get_info(ring, seg, offset & ~MASK,
            &ent->paddr,
            &ent->ptr, &ent->cb, &ent->s);
}
if (ent->ptr != NULL) {
    /* Cache/Memory */
    *(ent->ptr + (offset & MASK)) = val;
} else if (ent->cb != NULL) {
    /* I/O */
    ent->cb(ent->s, ent->paddr + (offset & mask), val);
} else {
    /* Seg-Fault, Watchpoint, Page-Fault, ... */
    store_slow(...);
}
...
```



Test auf geschützte Code-Seiten (JIT) entsprechend JIT-Kapitel auch noch integrierbar.

Voraussetzung:

Zwischen den Bits, die Cache-Zeile angeben, und den Bits, die das Alignment ausmachen, muss noch mindestens ein Bit übrig sein.

(Ist i.A. der Fall.)



Checks sind für die drei möglichen Speicherzugriffe (Execute, Read, Write) unterschiedlich: Z.B.

- Execute-Only Segmente,
- Read-Only Segmente,
- Execute-Only Seiten,
- Read-Only Seiten,
- Alignment-Checks nur für Daten-Zugriffe
- nur Ausführen/Lesen bei ROM-Bereichen
- ...

=> Execute/Load/Store-Funktionen nutzen jeweils eigene Zeiger bzw. Callbacks für gleiche Speicheradressen.



Da beim Code-Ausführen bzw. Daten-Lesen/-Schreiben meist unterschiedliche Speicherbereiche genutzt werden, komplett getrennte Caches:

```
struct entry {
    uint32_t offset;
    uint32_t paddr;
    uint8_t *ptr;
    void (*cb)(void *s, uint32_t paddr, uint8_t val);
    void *s;
};

static struct entry execute[4][1][EXECUTE_COUNT];
static struct entry read[4][6][READ_COUNT];
static struct entry write[4][6][WRITE_COUNT];
```



get_info-Funktion kann z.B.

- Platz im Cache suchen/freimachen
- Cache füllen
- Cache-LRU-Bits setzen
- Cache-Misses zählen (Hits nur indirekt)
- MMU-„Accessed“- und „Dirty“-Bits setzen
- ...

und muss die entsprechende Info zurückliefern.



Ergebnis:

Mit nur **drei** Speicherzugriffen

- Test des im Cache eingetragenen Offsets
- Auslesen des im Cache eingetragenen Zeigers
- Nutzen des Zeigers

kann ein **kompletter** Speicherhierarchie-**Speicher**-Zugriff emuliert werden!

Entsprechend: Speicherhierarchie-**I/O**-Zugriff (Auslesen und Nutzen des Callbacks).



Voraussetzungen:

alle Checks für den Speicherbereich fallen positiv aus:

- keine Segmentierungsverletzung
- keine Alignment-Verletzung
- kein aktiver Watchpoint in dem Bereich
- kein Page-Fault
- ...

Dies ist der Normalfall in allen(!) Betriebssystemen!

