Fortgeschrittene Speicherverwaltung

Translation Lookaside Buffer

Prof. Dr.-Ing. Andreas Heil

v1.0.0

Lernziele und Kompetenzen

Bisher Gelernt

- Paging ermöglicht es den Speicher in gleichgroße Abschnitte einzuteilen und diese zu adressieren
- Für Paging werden Paging Tables im Hauptspeicher vorgehalten, die sehr groß werden können
- Jeder Speicherzugriff (z.B. Variable laden oder schreiben, Instruktionen laden) benötigt für das Mapping der Adresse einen zusätzlichen Zugriff auf den Hauptspeicher
- Konsequenz?
    - Vermutlich werden wir durch den zusätzlichen Speicherzugriff langsamer…

Lösungsansatz: Hardware Support

- Um schneller zu werden, muss auch hier das Betriebssystem wieder durch die Hardware unterstützt werden
  - Nutzung eines sog. Translation Lookaside Buffers
  - Teil der MMU, also der Hardware bzw. der CPU
  - Vereinfacht: Ein Cache, in dem häufig genutzte Address-Mappings von virtuell zu physikalisch zwischengespeichert werden
- Bei jedem Speicherzugriff wird zuerst der TLB geprüft, ob das Mapping dort zwischengespeichert ist

Basisaligorithmus - Pseudocode

```
Ermittle Virtual Page Number (VPN) aus virtueller Adresse
Wenn VPN in TLB enthalten {
  Ermittle Page Frame Number (PFN)
  Verbinde pyhsikalische Adresse mit Offset auf virtueller Adresse
  Greife auf Speicher zu
}
Sonst {                                          //VPN nicht in TLB enthalten
  Greife auf Page Table zu und ermittle Mapping //das ist teuer
  Aktualisiere den TLB
  Wiederhole den Zugriff auf die TLB             //geht jetzt schnell
}
```

TLB und Implikationen für die Entwicklung

- Der TLB geht davon aus, dass Mappings im Cache gefunden werden
- Dadurch gibt es wenig Overhead, da der TLB schnell mit der CPU kommunizieren kann (im Gegensatz zum Hauptspeicher)
- Wenn oft Pages genutzt werden, deren Mappings nicht im TLB vorliegen, wird der Prozess merklich langsamer, da oft in den Hauptspeicher zugegriffen wird
- Bei den meisten modernen Sprachen macht man sich hier keinen Kopf aber auch hier gilt das Lokalitätsprinzip
- Problem, wenn oft auf Daten zugegriffen wird, die das * Laden von nicht »gecachten« Mappings erfordern
- Wie könnte man dem beim Programmieren entgegenwirken?

TLB Misses - Wer ist dafür zuständig?

- TLB Miss: Das Mapping liegt nicht im Cache
- Frühere Rechnerarchitekturen (complex-instruction set computers, abk. CISC) mussten wissen wo sich die Page Table befand und wie diese aussieht
- Bei einem Miss hat die Hardware nach dem korrekten Eintrag gesucht und den TLB aktualisiert
- Moderne Systeme (reduced-instruction set computers, abk. RISC) werfen eine Exception und ein Trap Handler startet im privilegierten Modus, der im Betriebssystem ausgeführt wird
- Nachdem mit privilegierten Operationen der TLB aktualisiert wurde, kehrt das Betriebssystem aus der Trap zurück

TLB Misses - Hinweis

- **Achtung**: Anders als bei anderen Traps, macht der Prozess nicht da weiter wo er aufgehört hat, sondern muss die letzte Instruktion (Speicherzugriff) wiederholen!
- Vorteil: Das Betriebssystem kann eine beliebige Datenstruktur für die Page Table nutzen und die Hardware wird einfacher (=günstiger)

Exkurs - Aber, der x86 iost doch CISC!?

- Eine der älteren CISC-Rechnerarchietekturen ist übrigens die x86-Architektur
- Tatsächlich ist heute die x86-Architektur die einzige nennenswerte Architektur, die noch einen CISC-Ansatz verfolgt
- Vorteile der x86 Architektur überwiegen auf vielen anderen Ebenen

Lesen!!1!!1: RISC vs. CISC:

- https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

TLB im Detail

- TLBs sind z.B. voll assoziativ (engl. fully associative), d.h. jedes Mapping kann überall im TLB stehen
- Hardware durchsucht den gesamten TLB (das wird sogar parallel gemacht)
- Ein typischer TLB-Eintrag ist wie folgt aufgebaut (wir kümmern uns gleich um die »weiteren Bits«):

TLB Bits

- Valid Bit: Der Eintrag im TLB hat ein gültiges Mapping
- Protection Bits: Wie kann/darf auf die Page zugegriffen werden
- Read/Execute: Heap Pages werden mit read und write gekennzeichnet
- Dirty Bit: s.vorher
- …

Context Switches (Sie erinnern sich hoffentlich...)

- Mappings in der TLB von virtuell zu physikalisch gelten nur für einen einzelnen Prozess
- Beim Context-Switch ist seitens Hardware und Betriebssystem darauf zu achten, dass der Prozess der laufen soll, nicht versehentlich auf Mappings des zuvor laufenden Prozesses zugreift

Context Switch Beispiel

- Beispiel: Prozess 1 mappt VPN 10 au 100 und Prozess 2 mappt VPN 10 auf 170...
  Was geschieht wenn Prozess 2 auf VPN 10 zugreifen will?

VPN	PFN	valid	prot
10	100	1	rwx
---	---	0	---
10	170	1	rwx
---	---	0	---

Variante 1

- TLB löschen (engl. flushen), z.B. wenn sich das Base Register ändert oder beim `sbrk` SysCall-Aufruf (vgl. o.)
- Bei vielen Wechseln, muss der Cache immer wieder aufgebaut werden (typisches Cache-Problem)
- Damit verbunden sind Performance-Einbußen

VPN	PFN	valid	prot
10	100	1	rwx
---	---	0	---
10	170	1	rwx
---	---	0	---

Variante 2

- Nutzen eines sog. *Address Space Identifiers*
- Analog zur PID, nur mit weniger Bits (8 Bit anstelle 32)

VPN	PFN	valid	prot
10	100	1	rwx
---	---	0	---
10	170	1	rwx
---	---	0	---

Replacement Policy

- Typisches Problem in einem Cache
- Irgendwann ist der Cache voll und alte Einträge müssen gegen neue ausgetauscht werden
  *`Hier stellt sich die Frage, WIE!?!!
- Ein möglicher Ansatz: Least Recently Used (LRU), zuvor schon erwähnt
  - Grundlegende Idee: Einträge, die schon lange nicht mehr angefasst werden, werden mit hoher Wahrscheinlichkeit demnächst auch nicht benötigt
  - Das verhält sich so ähnliche wie mit den Sachen in Kisten bei Ihnen im Keller…
- Es gibt allerdings noch mehr Ansätze, bis hin zu einer reinen Zufallsauswahl

Referenzen

Bildnachweise