Universität Bamberg

GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

Vorlesung 15:
Virtueller Speicher

Literatur:
Arpaci-Dusseau [1]
Kap. 18–20

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Virtueller Speicher

- Die direkte Verwendung von physischem Hauptspeicher ist problematisch
  - Programme müssen an **unterschiedliche Speicherbereiche gelinkt werden**, um **sich nicht gegenseitig zu überschreiben**, wenn sie gleichzeitig laufen (= im Speicher sein) sollen
  - Schutz des phys. Speichers ist meist nicht vorhanden oder nur sehr grobgranular und unflexibel
  - Programme, die größer als der verfügbare Hauptspeicher sind, sind nur schwer handhabbar (→ Overlays)

- **Lösung:**
  - Komponente, die die von der Software erzeugten Adressen in Adressen, die zur Ansteuerung des Hauptspeichers verwendet werden, **übersetzt**
  - Erzeugen der Illusion, dass jeder Prozess den gesamten (virtuellen) Adressraum für sich alleine besitzt

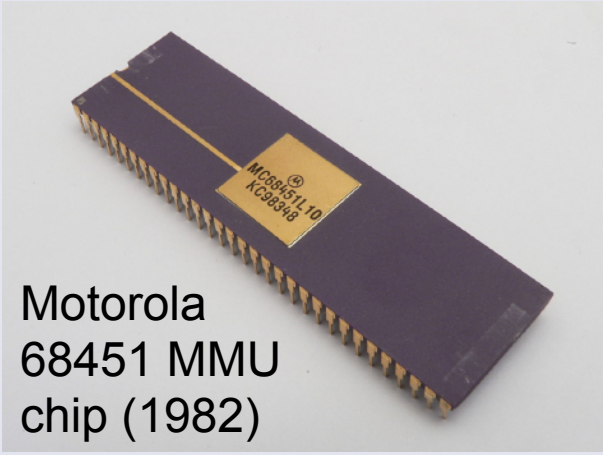
Virtueller Speicher

- **Zusätzlicher Vorteil:**
  Entkoppeln der Speicheranforderungen von verfügbarer Menge Hauptspeicher
  - Prozesse greifen nicht mit der selben Häufigkeit auf alle Speicheradressen zu
  - Bestimmte Instruktionen werden nur selten verwendet (ausgeführt) oder auch überhaupt nicht (z.B. Fehlerbehandlung, wenn keiner auftritt)
  - Bestimmte Datenstrukturen werden nicht komplett genutzt
  - Prozesse können mehr Speicher verwenden, als Hauptspeicher zur Verfügung steht

- **Idee:**
  - Illusion eines großen Hauptspeichers erzeugen
  - Aktuell benutzte Speicherbereiche in Hauptspeicher halten
  - Zugriffe auf Bereiche, die gerade nicht im Hauptspeicher sind, abfangen
    - …und bei Bedarf laden ("einlagern")
  - Aktuell nicht benutzte Bereiche auslagern

MMU – Memory Management Unit

- Idee: Abfangen von ***virtuellen Adressen*** von der CPU
  - MMU prüft auf **zulässige** Adressen
  - Übersetzt zulässige Adressen in physische Adressen zur Ansteuerung des Hauptspeichers über **Übersetzungstabelle**
- **Problem:** Tabelle mit Einträgen für einzelne Adressen ist sehr groß
  - Speicher wird in **Seiten** (***pages***) mit identischer Größe aufgeteilt (Zweierpotenz)
  - Selbe Übersetzung für alle Adressen einer Seite:
    Seitentabelle (***page table***)
- MMUs waren ursprünglich separate ICs zwischen CPU und RAM
  - Heute: Höhere Integration
    ➛ Teil des CPU-Chips

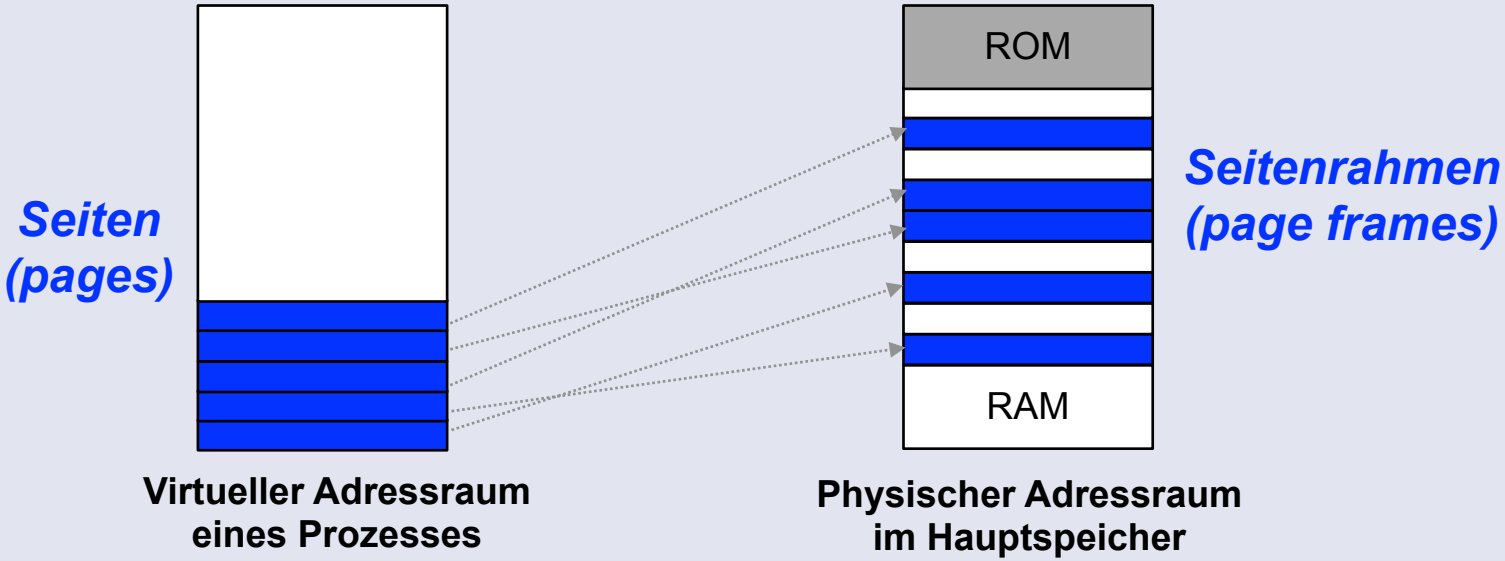


Motorola 68451 MMU chip (1982)

[Wikimedia by David Monniaux, CC BY-SA 3.0]

Paging – seitenbasierter virtueller Speicher

- Der **virtuelle Adressraum** ist in **Seiten (pages)** gleicher Größe aufgeteilt
  - Seiten können an beliebigen Positionen im physischen Speicher liegen
    - …beginnend bei einem Vielfachen der Seitengröße
  - Ermöglicht einfachere Speicherallokation und -austausch



Seitenersetzung

- Was geschieht, wenn kein Seitenrahmen verfügbar ist, um eine Speicheranforderung zu erfüllen?
  - (Mindestens) eine Seite muss aus dem Hauptspeicher verdrängt werden, um Platz für die neue Seite zu schaffen!
  - Zunächst werden Seiten mit nicht geändertem Inhalt gewählt (durch das *dirty*-Bit im Seitentabelleneintrag angezeigt)
  - Verdrängung einer Seite erfordert, dass deren Inhalte auf die Festplatte ausgelagert werden, wenn geänderte Inhalte vorliegen
- **Ablauf:**
  - ***Seitenfehler***: ***trap*** in das Betriebssystem
  - *Verdrängen (**Auslagern**)* eines Seitenrahmens, falls kein freier verfügbar
  - ***Einlagern*** der angeforderten Seite
  - Wiederholung des fehlgeschlagenen Speicherzugriffs
- **Problem:**
  - Welche Seite sollte verdrängt werden (das “Opfer”)?

Ersetzungsstrategien

- Wir betrachten *Ersetzungsstrategien* und ihre Auswirkung auf ***Speicherzugriffsfolgen***
- *Zugriffsfolge:*
  - Folge von Seitennummern, die das ***Speicherzugriffsverhalten*** eines Prozesses beschreiben
  - Bestimmung der *Zugriffsfolgen* z.B. durch Aufzeichnung der von einem Prozess erzeugten Lade/Schreiboperationen
    - Reduktion der Aufzeichnung auf Seitennummern
    - Zusammenfassen aufeinanderfolgender Zugriffe zu einem
  - Beispiel für eine Zugriffsfolge:
    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

First in, first out

- Ersetzung der *ältesten Seite*
- Benötigte Information:
  - "Alter" = Zeitpunkt der Einlagerung für jeden Seitenrahmen

- Ersetzungsreihenfolge (9 Einlagerungen):

Zugriffsfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Rahmen 1	**1**	1	1	**4**	4	4	**5**	5	5	5	5	5
	Rahmen 2		**2**	2	2	**1**	1	1	1	1	**3**	3	3
	Rahmen 3			**3**	3	3	**2**	2	2	2	2	**4**	4
Zustand ("Alter" des Seitenrahmens)	Rahmen 1	0	1	2	0	1	2	0	1	2	3	4	5
	Rahmen 2	>	0	1	2	0	1	2	3	4	0	1	2
	Rahmen 3	>	>	0	1	2	0	1	2	3	4	0	1

Optimale Ersetzungsstrategie

- Betrachtung des *Vorwärtsabstandes*
  - Zeit, bis der nächste Zugriff auf die jeweilige Seite erfolgt
- Strategie OPT (auch: MIN) ist optimal (für eine festgelegte Anzahl an Seiten): minimale Anzahl von Ein-/Auslagerungen (hier: 7)
  - “Ersetze immer die Seite, deren Zugriff am weitesten in der Zukunft liegt (mit dem größten *Vorwärtsabstand*)"

Zugriffsfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Rahmen 1	**1**	1	1	1	1	1	1	1	1	**3**	**4**	4
	Rahmen 2		**2**	2	2	2	2	2	2	2	2	2	2
	Rahmen 3			**3**	**4**	4	4	**5**	5	5	5	5	5
Zustand ("Alter" des Seitenrahmens)	Rahmen 1	4	3	2	1	3	2	1	>	>	>	>	>
	Rahmen 2	>	4	3	2	1	3	2	1	>	>	>	>
	Rahmen 3	>	>	7	7	6	5	5	4	3	2	1	>

First in, first out

- Größerer Hauptspeicher: jetzt 4 Seitenrahmen (10 Einlagerungen)
- FIFO-Anomalie (Béládys Anomalie, 1969)

Zugriffsfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Rahmen 1	**1**	1	1	1	1	1	**5**	5	5	5	**4**	4
	Rahmen 2		**2**	2	2	2	2	2	**1**	1	1	1	**5**
	Rahmen 3			**3**	3	3	3	3	3	**2**	2	2	2
	Rahmen 4				**4**	4	4	4	4	4	**3**	3	3
Zustand ("Alter" des Seitenrahmens)	Rahmen 1	0	1	2	3	4	5	0	1	2	3	0	1
	Rahmen 2	>	0	1	2	3	4	5	0	1	2	3	0
	Rahmen 3	>	>	0	1	2	3	4	5	0	1	2	3
	Rahmen 4	>	>	>	0	1	2	3	4	5	0	1	2

Optimale Ersetzungsstrategie

- Größerer Hauptspeicher: jetzt 4 Seitenrahmen (6 Einlagerungen)
  - ***keine Anomalie***

Zugriffsfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Rahmen 1	**1**	1	1	1	1	1	1	1	1	1	**4**	4
	Rahmen 2		**2**	2	2	2	2	2	2	2	2	2	2
	Rahmen 3			**3**	3	3	3	3	3	3	3	3	3
	Rahmen 4				**4**	4	4	**5**	5	5	5	5	5
Zustand ("Alter" des Seitenrahmens)	Rahmen 1	4	3	2	1	3	2	1	>	>	>	>	>
	Rahmen 2	>	4	3	2	1	3	2	1	>	>	>	>
	Rahmen 3	>	>	7	6	5	4	3	2	1	>	>	>
	Rahmen 4	>	>	>	7	6	5	5	4	3	2	1	>

Optimale Ersetzungsstrategie

- Die Implementierung von OPT ist praktisch unmöglich
  - …da wir *im Vorhinein* die Zugriffsfolge kennen müssten!
  - OPT ist aber nützlich, um *Strategie zu vergleichen*

- **Gesucht:** Strategien, die möglichst nahe an OPT liegen
  - z.B. ***Least Recently Used*** (LRU)

Least recently used (LRU)

- Betrachtung des *Rückwärtsabstandes*
  - Zeit, die seit letztem Zugriff auf die Seite vergangen ist
- LRU-Strategie (10 Einlagerungen)
  - “Ersetze die Seite, deren letzter Zugriff am weitesten in der Vergangenheit liegt, also mit größtem *Rückwärtsabstand*!”

Zugriffsfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Rahmen 1	**1**	1	1	**4**	4	4	**5**	5	5	**3**	3	3
	Rahmen 2		**2**	2	2	**1**	1	1	1	1	1	**4**	4
	Rahmen 3			**3**	3	3	**2**	2	2	2	2	2	**5**
Zustand ("Alter" des Seitenrahmens)	Rahmen 1	0	1	2	0	1	2	0	1	2	0	1	2
	Rahmen 2	>	0	1	2	0	1	2	0	1	2	0	1
	Rahmen 3	>	>	0	1	2	0	1	2	0	1	2	0

Least recently used (LRU)

- Größerer Hauptspeicher: jetzt 4 Seitenrahmen (8 Einlagerungen)

Zugriffsfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Rahmen 1	**1**	1	1	1	1	1	1	1	1	1	1	**5**
	Rahmen 2		**2**	2	2	2	2	2	2	2	2	2	2
	Rahmen 3			**3**	3	3	3	**5**	5	5	5	**4**	4
	Rahmen 4				**4**	4	4	4	4	4	**3**	3	3
Zustand ("Alter" des Seitenrahmens)	Rahmen 1	0	1	2	3	0	1	2	0	1	2	3	0
	Rahmen 2	>	0	1	2	3	0	1	2	0	1	2	3
	Rahmen 3	>	>	0	1	2	3	0	1	2	3	0	1
	Rahmen 4	>	>	>	0	1	2	3	4	5	0	1	2

Least recently used (LRU)

- Keine Anomalie
  - Allgemein: es gibt eine Klasse ein Algorithmen (**Stack-algorithmen**), die keine Anomalie aufweiten:
    - Für Stackalgorithmen mit *k* Seitenrahmen gilt:
      Zu jedem Zeitpunkt ist eine Teilmenge der Seiten eingelagert, die zum selben Zeitpunkt auch in einem System mit *k+1* Seitenrahmen eingelagert wären!
    - LRU: die *k* zuletzt verwendeten Seiten sind eingelagert
    - OPT: die *k* Seiten sind die Seiten, auf die als Nächste zugegriffen wird
- **Problem**
  - Implementierung von LRU benötigt Hardwareunterstützung
  - Jeder Speicherzugriff muss betrachtet werden

Least recently used (LRU)

- Naive Idee: Hardwareunterstützung durch *Zähler*
  - CPU implementiert einen Zähler, der mit jedem Speicherzugriff inkrementiert wird
  - Bei jedem Zugriff wird der aktuelle Wert des Zählers in den entsprechenden Seitendeskriptor geschrieben
  - Wähle die Seite mit kleinstem Zählerwert (also die älteste) aus
    ➛ ***benötigt eine Suche!***

- Großer Implementierungsaufwand
  - Viele zusätzliche Speicherzugriffe benötigt
  - Viel zusätzlicher Speicher benötigt (für Zählerwerte)
  - Suche des Minimums muss in der Funktion zur Seitenfehlerbehandlung (*page fault handler*) stattfinden

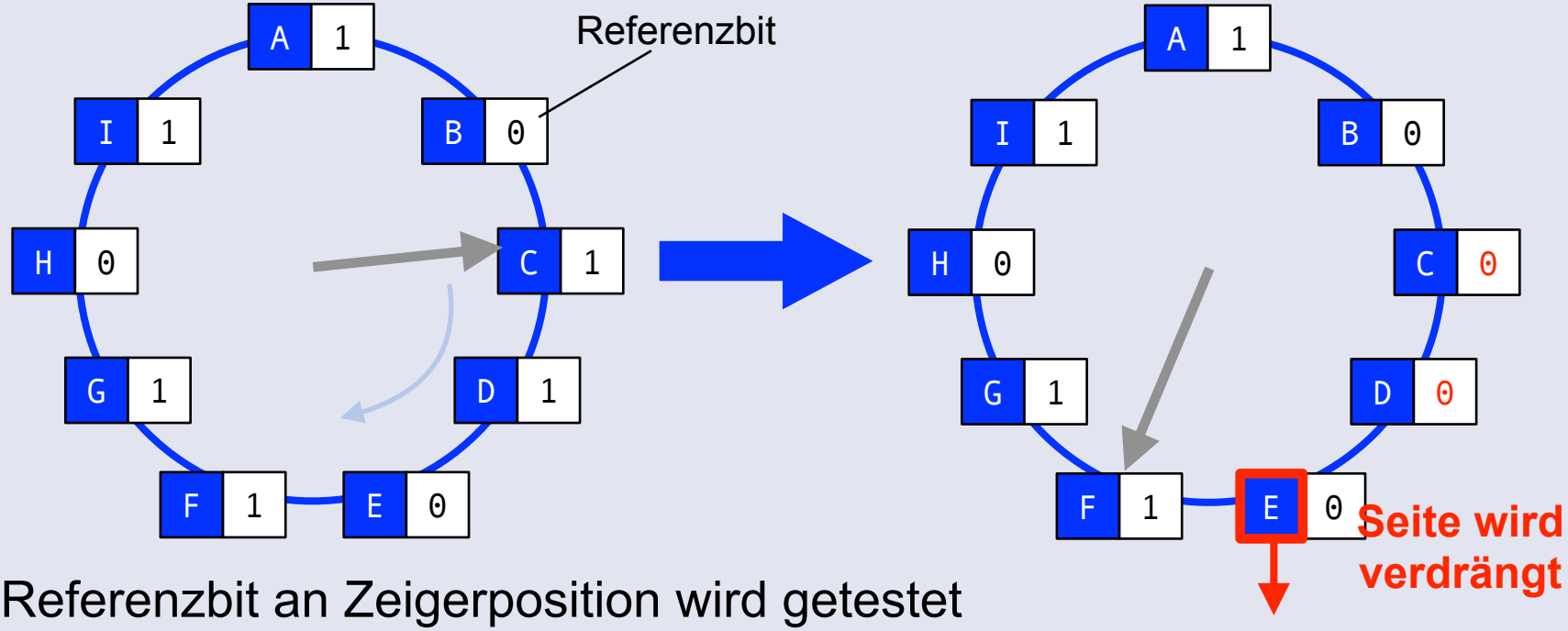
Second chance (Uhralgorithmus)

- Funktionierender Ansatz: Verwendung von ***Referenzbits***
  - Referenzbit im Seitentabelleneintrag wird automatisch von der Hardware gesetzt, wenn auf eine Seite zugegriffen wird
  - Einfacher implementierbar
  - Erfordert weniger zusätzliche Speicherzugriffe
- Moderne Prozessoren/MMUs verwenden Referenzbits (z.B. unter dem Namen “access bit” bei x86)
- Ziel: Annäherung an LRU
  - Referenzbit einer neu eingelagerten Seite initial auf 1 gesetzt
  - Wenn ein “Opfer” zur Auslagerung benötigt wird, werden Referenzbits in dieser Reihenfolge betrachtet:
    - wenn Referenzbit = 1, setze es auf 0 (*second chance*)
    - wenn Referenzbit = 0, verdränge diese Seite!

Second chance (Uhr)

Universität Bamberg

- Implementierung durch rotierenden Zeiger (Uhr/*clock*)



- Referenzbit an Zeigerposition wird getestet
  - wenn Referenzbit = 1: löschen (auf 0 setzen)
  - wenn Referenzbit = 0: wir haben verdrängbare Seite gefunden!
  - Zeiger “tickt” weiter: wenn keine Seite gefunden wurde, von vorne anfangen
- Wenn alle Referenzbits = 1 sind, dann ist second chance eine FIFO

Second chance (Uhr)

- Zugriffsfolge mit 3 Seitenrahmen:
  9 Einlagerungen

Zugriffsfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Rahmen 1	**1**	1	1	**4**	4	4	**5**	5	5	5	5	5
	Rahmen 2		**2**	2	2	**1**	1	1	1	1	**3**	3	3
	Rahmen 3			**3**	3	3	**2**	2	2	2	2	**4**	4
Zustand (Referenzbits)	Rahmen 1	1	1	**1**	1	1	**1**	1	1	1	0	**0**	**1**
	Rahmen 2	**0**	1	1	**0**	1	1	**0**	**1**	**1**	1	1	1
	Rahmen 3	0	**0**	1	0	**0**	1	0	0	1	**0**	1	1
	Zeigerposit.	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**2**	**2**	**3**	**1**	**1**

Second chance (Uhr)

- Größerer Hauptspeicher (4 Seitenrahmen):
  10 Einlagerungen

Zugriffsfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Rahmen 1	**1**	1	1	**1**	1	1	**5**	5	5	5	**4**	4
	Rahmen 2		**2**	2	2	2	2	2	**1**	1	1	1	**5**
	Rahmen 3			**3**	3	3	3	3	3	**2**	2	2	2
	Rahmen 4				**4**	4	4	4	4	4	**3**	3	3
Zustand (Referenzbits)	Rahmen 1	1	1	1	**1**	**1**	**1**	1	1	1	**1**	1	1
	Rahmen 2	**0**	1	1	1	1	1	**0**	1	1	1	**0**	1
	Rahmen 3	0	**0**	1	1	1	1	0	**0**	1	1	0	**0**
	Rahmen 4	0	0	**0**	1	1	1	0	0	**0**	1	0	0
	Zeigerposit.	**2**	**3**	**4**	**1**	**1**	**1**	**2**	**3**	**4**	**1**	**2**	**3**

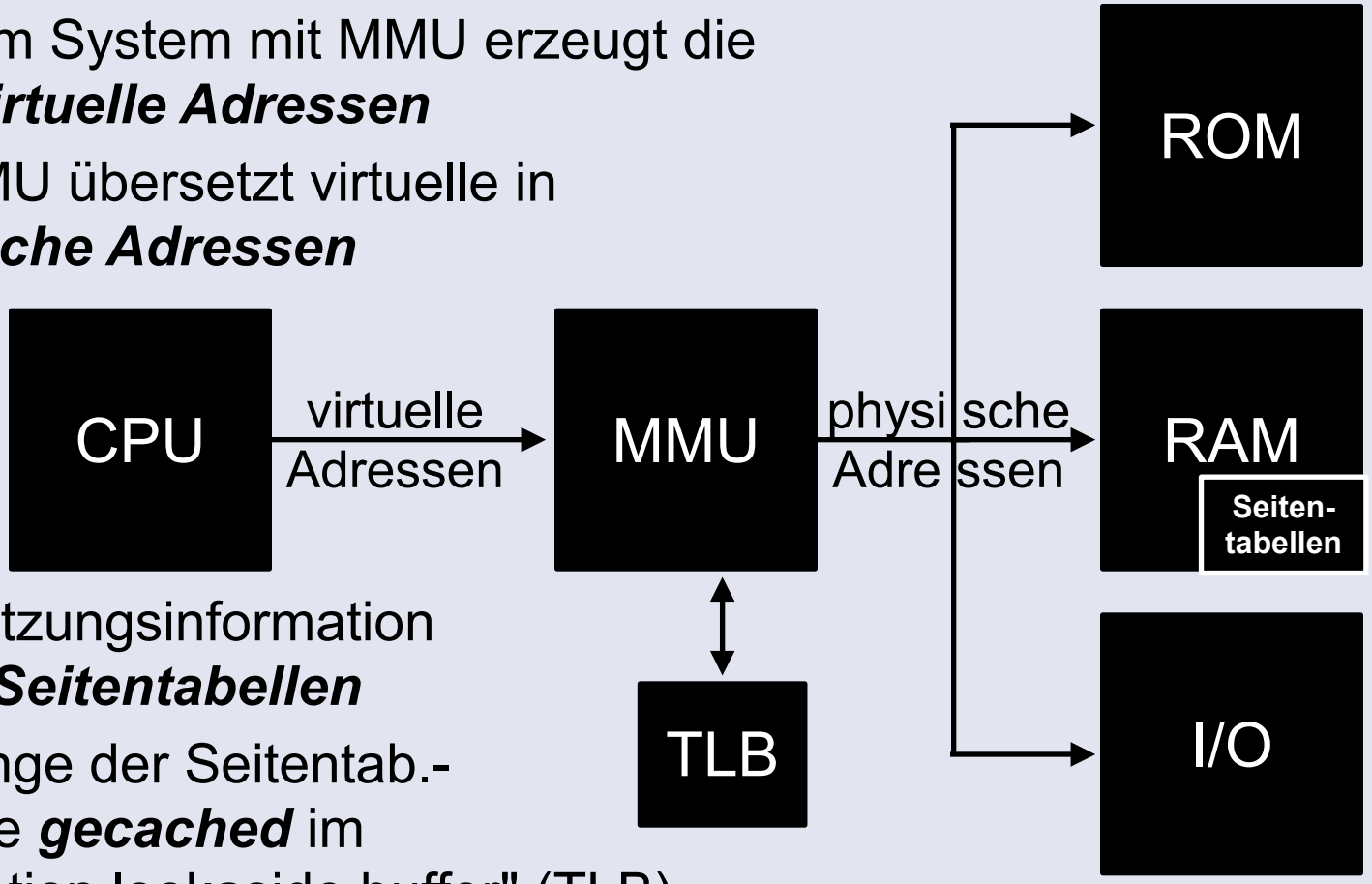
Second chance (Uhr)

- Second chance kann auch zu FIFO "degenerieren"
  - Wenn alle Referenzbits = 1 sind, ist dies FIFO-Reihenfolge
- Gewöhnlich liegt second change aber nahe an LRU

- Erweiterung
  - Modifikationsbit kann zusätzlich betrachtet werden (*dirty bit*)
  - Drei Klassen des Tupels (Referenzbit, Modifikationsbit) :
    (0,0), (1,0) und (1,1)
  - Suche nach der “niedrigsten” Klasse (in macOS verwendet)

Struktur eines Computers mit MMU

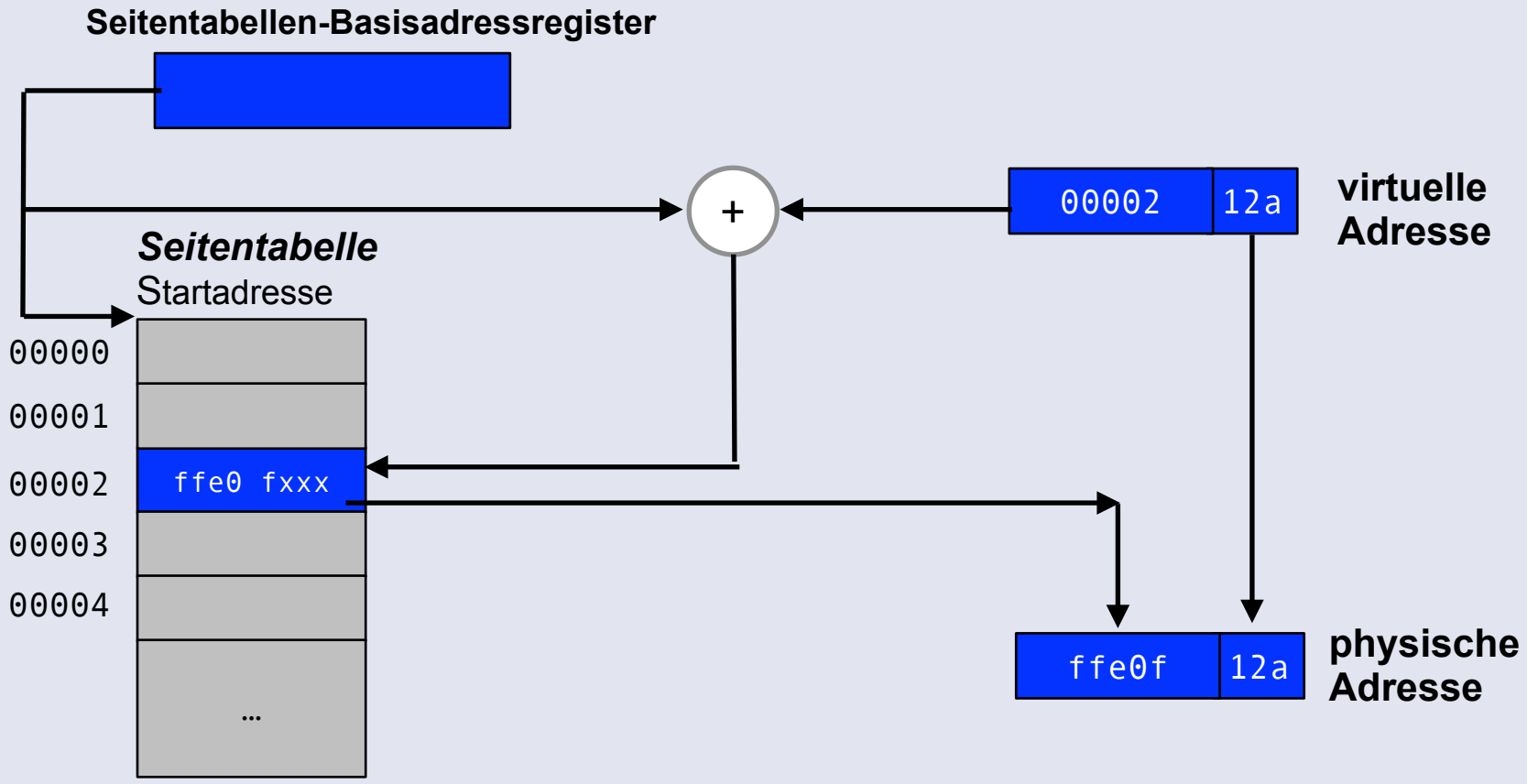
- Von Software in der CPU erzeugte Adressen sprechen nicht mehr direkt den Speicher (oder *memory mapped* Peripherie) an
- In einem System mit MMU erzeugt die CPU ***virtuelle Adressen***
- Die MMU übersetzt virtuelle in ***physische Adressen***



- Übersetzungsinformation in den ***Seitentabellen***
- Teilmenge der Seitentab.-einträge ***gecached*** im "translation lookaside buffer" (TLB)

MMU mit Seitentabellen

- Allgemein: eine Tabelle wird genutzt, um *Seitenadressen* in *Seitenrahmenadressen* zu übersetzen



MMU mit Seitentabellen (2)

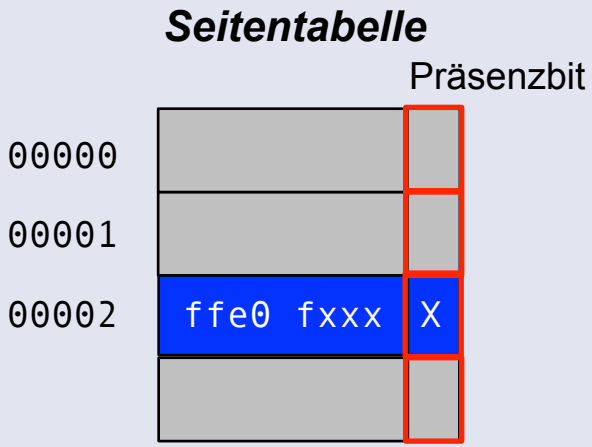
- Seitenbasierte Adressierung erzeugt ***interne Fragmentierung***
  - Die letzte Seite wird oft nicht vollständig genutzt
- Festlegung der Seitengröße
  - Kleine Seiten reduzieren interne Fragmentierung, erhöhen aber die Größe der Seitentabelle (und umgekehrt)
  - Übliche Seitengrößen: 512 Byte — 16384 Byte

- Seitentabellen sind groß, müssen im Hauptspeicher vorhanden sein
  - Das OS muss die ***Seitentabellen*** für jeden Prozess vor dem Start ***anlegen*** und muss diese während der Ausführung ***aktualisieren***
- Große Anzahl impliziter Seitenzugriffe erforderlich, um eine Adresse zu übersetzen

Metadaten der Seitentabelleneinträge

- Die physische Adresse alleine ist nicht ausreichend: Weitere Information sind notwendig, um virtuellen Speicher effizient zu nutzen

- Hardwareunterstützung
  - Wenn das ***Präsenzbit*** gesetzt ist, ist die Seite im Speicher vorhanden
  - Wenn das Präsenzbit nicht gesetzt wird, wird eine Exception ausgelöst (***Seitenfehler*** oder ***page fault***)



  - Der Exception-Handler (Teil des OS) kann nun das Einlagern der Seite vom Hintergrundspeicher (Swap-Partition oder Swapfile) anstoßen

Naive Seitentabellen

- Wir benötigen eine physische Adresse (**Seitenrahmen**, **page frame**) für jede virtuelle Speicherseite
- Wie viel Speicher benötigt die Seitentabelle selbst?

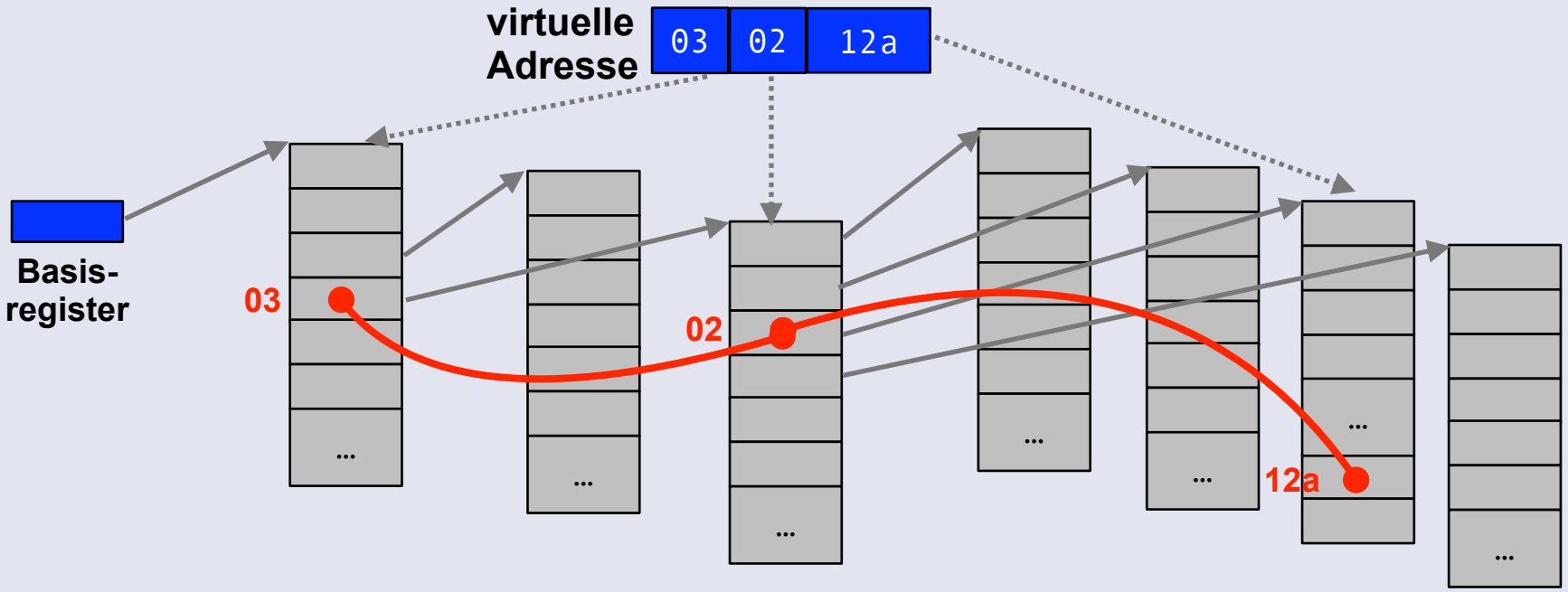
- Beispiel: 4 GB (32 Bit) Adressraum, 4 kB (2^{12} Bytes) Seitengröße
  $\Rightarrow 2^{32} / 2^{12} = 2^{20}$ (~1 Million) Einträge – pro Prozess!

  $\Rightarrow$ mit 4 Bytes pro Eintrag belegt die Seitentabelle 4 MB RAM!
- Die meisten Seitentabelleneinträge wären leer
  - …wenn ein Prozess nicht den größten Teil seines Adressraums von 4 GB ausnutzt

- Idee: verwende ***dünn besetzten Speicher*** für Seitentabellen

Mehrstufige Seitenadressierung

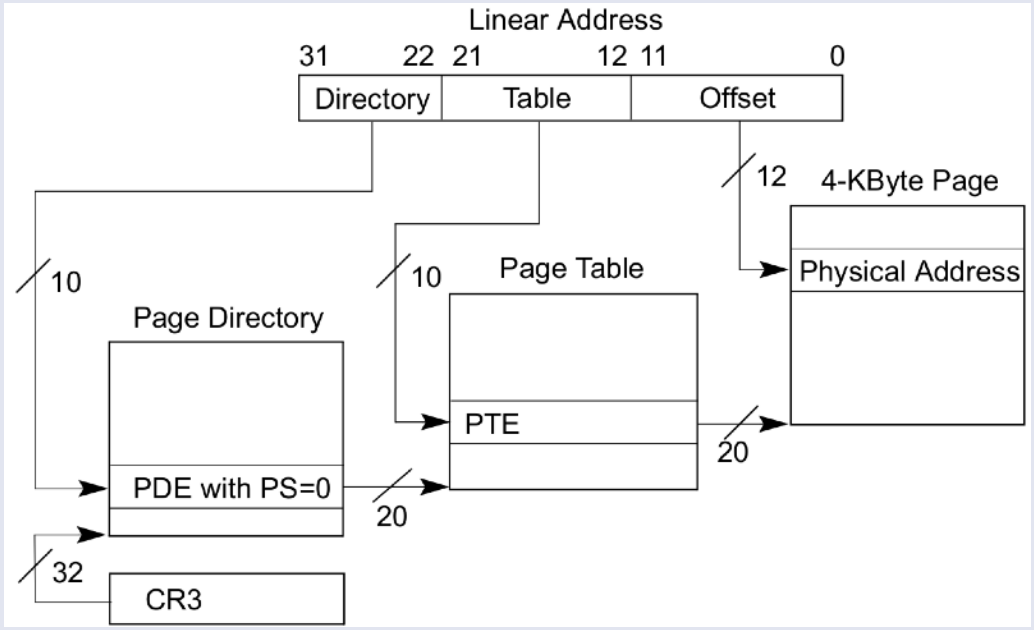
- Beispiel: zweistufige Seitenadressierung



- Präsenzbit existiert auch für alle Einträge auf höheren Stufen
  - Ermöglicht es, die Seitentabellen selbst auszulagern
  - Tabellen können bei Bedarf erzeugt werden – spart Speicher!
- Aber: noch mehr implizite Speicherzugriffe notwendig!

Hierarchische Seitentabellen

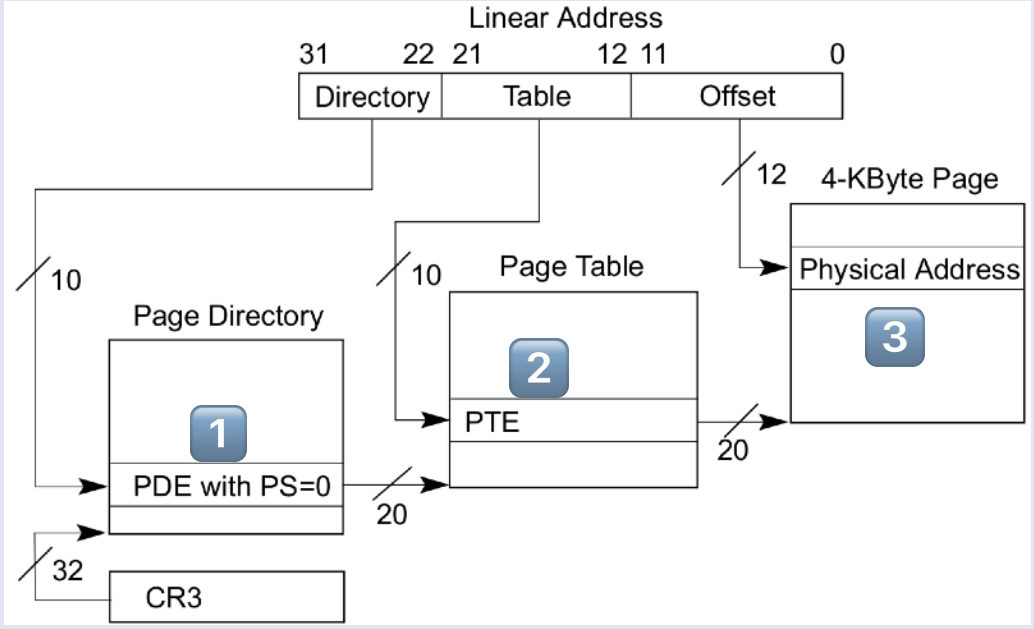
- Speicher in **Seiten** identischer Größe geteilt (Zweierpotenz)
- Die selbe Übersetzung für alle Adressen einer Seite anwenden: **Seitentabelle**
- Welche **Seitengröße** ermöglicht Flexibilität *und* Effizienz?
  - Mehrere kB: 4 kB=2^{12} Bytes (x86), 16 kB (Apple M1)
- Verwendung dünn besetzter mehrstufiger Seitentabellen
  ➛ Reduzierte Tabellengröße
- z.B. für 32 Bit x86:
  - Seitengröße:
    - 2^{12} = 4096 Bytes
  - Seitentabelle:
    - 2^{10} Seiteneinträge
  - Seitenverzeichnis:
    - 2^{10} Seitentabellen



Zweistufige Adressübersetzung

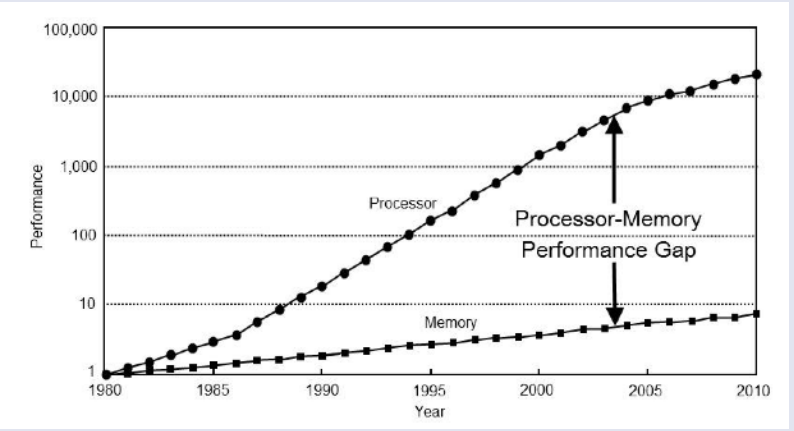
- Die MMU teilt die virtuelle (oder “lineare”) Adresse der CPU in drei Teile:
  - 10 Bit (31–22) Nummer des Seitenverzeichnisses (page directory entry, PDE)
  - 10 Bit (21–12) Nummer des Seitentabelleneintrags (page table entry, PTE)
  - 12 Bit (11–0) Seiten-***Offset*** innerhalb der angesprochenen Seite (wird ***nicht übersetzt***!)

- **Übersetzungsvorgang:**
  - Lies PDE-Eintrag aus Verz.: ➛ Adresse einer Seitentabelle
  - Lies PTE-Eintrag aus Verz.: ➛ physische Basisadresse der Speicherseite
  - Addiere Offset von der virtuellen Adresse (Bit 11–0), um die komplette physische Speicheradresse zu erhalten



Beschleunigung der Adressübersetzung

- Wo ist die Seitentabelle gespeichert?
- *Kann* mehrere MB groß sein
  ➛ passt nicht auf den CPU-Chip!
- Seitenverzeichnisse und Seiten-tabellen liegen im **Hauptspeicher**!

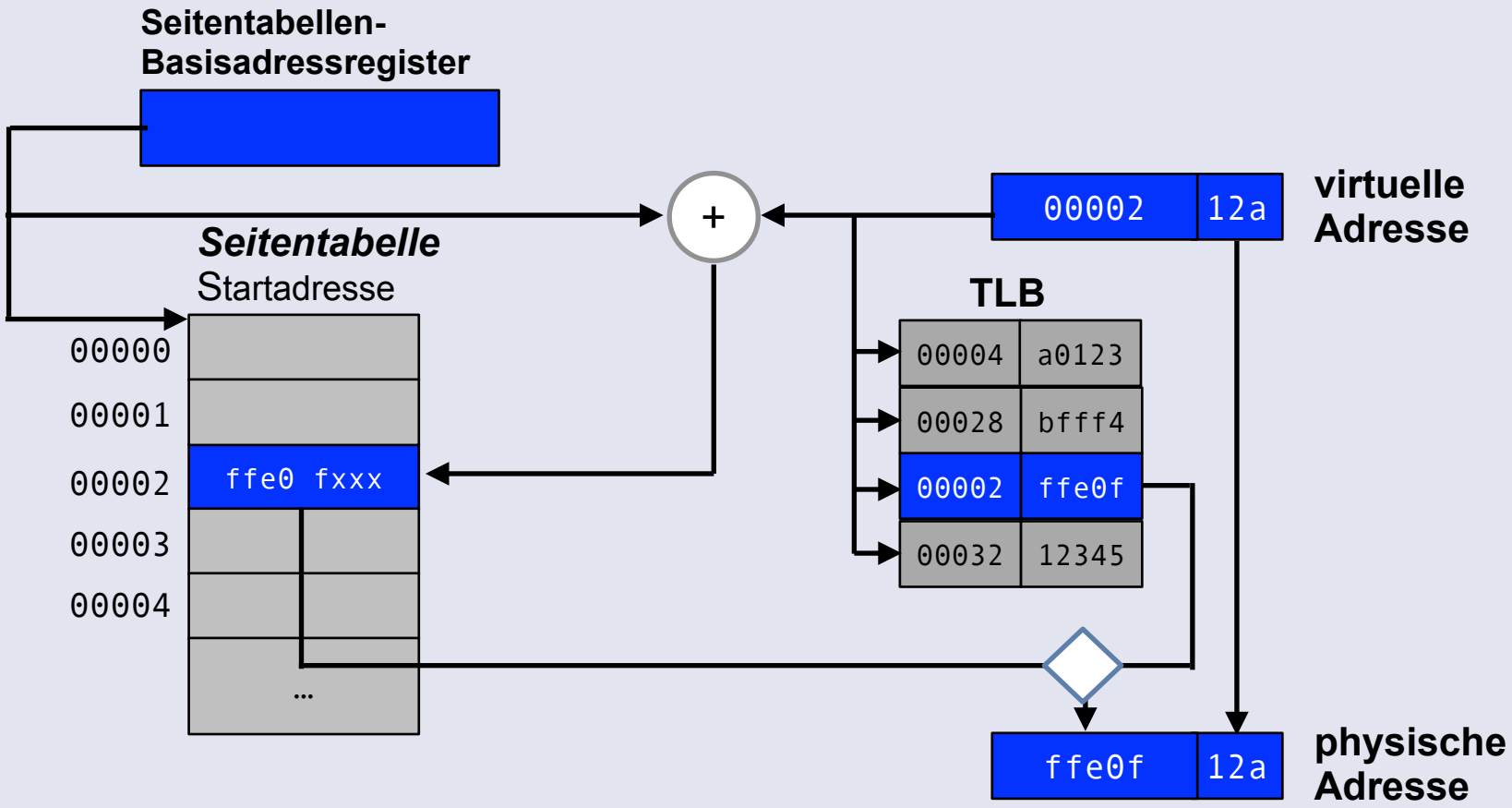


- Verwendung von virtueller Adressübersetzung erfordert **mehrere Hauptspeicherzugriffe** (z.B. zwei bis vier Stufen)!

- Gleiche Idee wie bei der Beschleunigung langsamer Hauptspeicherzugriffe: Verwendung eines **Caches**!
- Die MMU verwendet einen besonderen Cache auf dem CPU-Chip: den **Translation Lookaside Buffer** (TLB)
- Speichert die "meist" (häufigste? aktuellste?) verwendeten PTEs

Translation Lookaside Buffer (TLB)

- Schneller ***Cache***, in dem vor dem (möglichen) Nachschlagen in der Seitentabelle nachgesehen wird:



Translation Lookaside Buffer (TLB)

- Schneller Zugriff auf Adressübersetzung – Information im (voll assoziativen) TLB-Speicher
- TLB muss ***geleert (flush)*** werden, wenn ein Kontextwechsel stattfindet
- Wenn eine Übersetzung nicht im TLB-Cache enthalten ist, muss sie bei Bedarf eingetragen werden
  - Dazu wird ein existierender TLB-Eintrag *verdrängt*

- TLB-Größen:
  - Intel Core i7: 512 Einträge, Seitengröße 4 kB
  - UltraSPARC T2: Daten-TLB = 128, Code-TLB = 64, Seitengr. 8 kB
  - Allwinner/T-Head C906 RISC-V: 128-512 Einträge
  - Größere TLBs sind wegen Timingproblemen und hohen Kosten meist nicht realisierbar

Virtueller Speicher bei RISC-V

- Die Modi der virtuellen Adressübersetzung sind "sv**xx**"
  - "**xx**" ist eine Zahl, die die Anzahl der übersetzten Bits angibt
- 32-bit RISC-V-Systeme nutzen den sv32-Modus
  - zwei Übersetzungsstufen mit je 10 Bit Index (1024 Einträge), 12 Bit (4 kB) Seitengröße

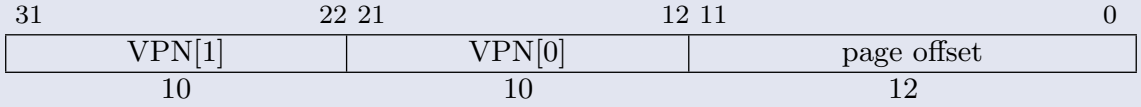


Figure 4.12: Sv32 virtual address.

- 64-bit RISC-V besitzt mehrere Optionen:
  - sv39, 48 und 57 aktuell definiert: drei Übersetzungsstufen

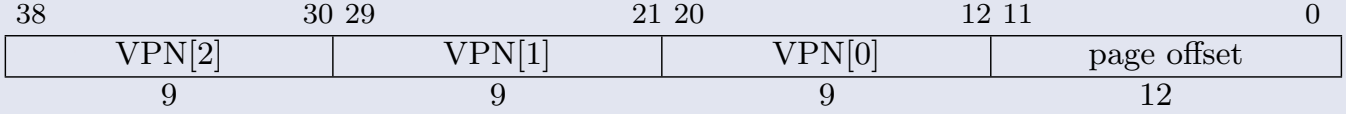


Figure 4.15: Sv39 virtual address.

Seitentabellen bei RISC-V

Universität Bamberg

Format der Seitentabelleneinträge definiert durch spezifischen sv-Modus:

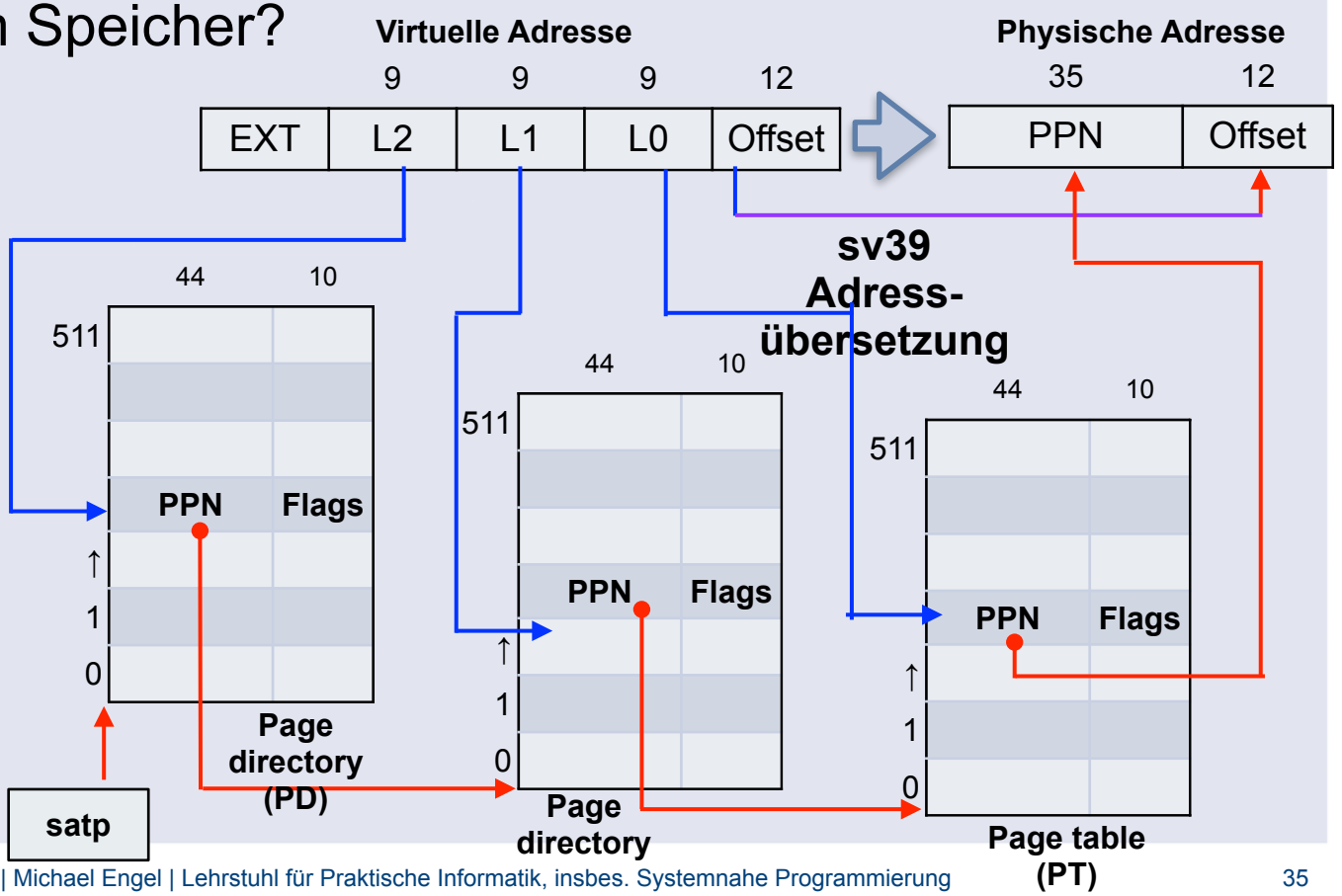
63	62–61	60–54	53–28	27–19	18–10	9–8	7	6	5	4	3	2	1	0
N	PBMT	*Reserved*	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	7	26	9	9	2	1	1	1	1	1	1	1	1

Figure 4.21: Sv39 page table entry.

- MMU ersetzt Bits 12–38 der virt. Adr. durch ***phys. Seitennummer*** (**PPN**)
  - 44 Bits für physische Seitennummer verfügbar
  - *größerer physischer als virtueller Adressraum möglich!*
- Metainformation:
  - **V**(alid): ist Eintrag gültig? *(einziges gesetztes Bit für page directories)*
  - Protection bits: **R**(ead), **W**(rite), (e)**X**(ecute) Zugriffsrechte für Seite
  - **U**(ser): Eintrag gültig für U-Modus
  - **G**(lobal) bit: Eintrag gültig in allen Adressräumen
  - **A**(ccessed): auf Seite wurde seit Löschen des A-Bits zugegriffen
  - **D**(irty): Seite wurde beschrieben, seit D-Bit gelöscht wurde

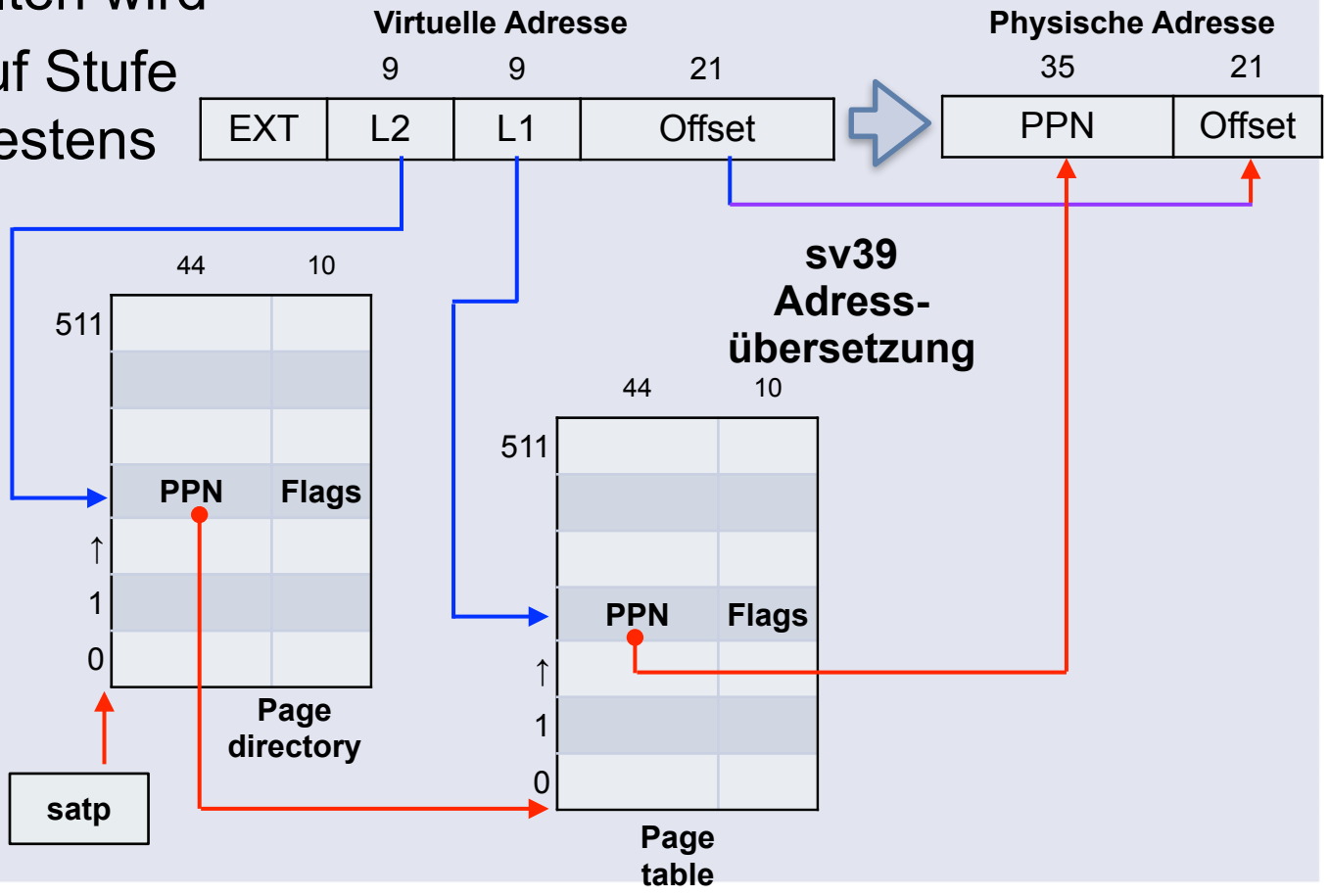
Virtueller Speicher bei RISC-V

- Virtuelle Adressübersetzung nur in S- und U-Modus
  - "Normale" RISC-V-Betriebssysteme laufen im S-Modus
- Wie findet die CPU ihre Seitentabellen im Speicher?
  - `satp` CSR "*supervisor address translation pointer*"
  - Zeigt auf Start der **phys. Seite** der Tabelle auf Stufe 1



Große Seiten bei RISC-V

- Bei dreistufiger Tabelle sind Seiten 2^{12}=4096 Bytes groß
- Man kann **große Seiten** erzeugen, indem die Übersetzung vorzeitig angehalten wird
- Wenn in **flags** auf Stufe L2 oder L1 mindestens ein Bit aus R, W, or X gesetzt ist, dann stoppt der **page table walk** bei dieser Stufe
- **Stop bei L1:** Seitengröße = 4096*512 = 2MB



Virtueller Speicher bei RISC-V

- Virtuelle Adressübersetzung standardmäßig nicht aktiv
  - Aktiviert durch Schreiben in `satp` CSR

Figure 4.12: RV64 Supervisor address translation and protection register `satp`, for MODE values Sv39 and Sv48.

- Bit 0–43: Physische Seitennummer der obersten Seitentabelle
- Bit 44–50: Adressraum-ID (welcher Prozess?)
- Bit 60–63: MMU-Modus

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved*
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.
10	*Sv57*	*Reserved for page-based 57-bit virtual addressing.*
11	*Sv64*	*Reserved for page-based 64-bit virtual addressing.*
12–15	—	*Reserved*

MMU-Modusbits

- Sicherstellen, dass Übersetzungen **synchronisiert** sind
  - `sfence.vma`-Instruktion nach Ändern von `satp`

RISC-V-Adressübersetzung: 3 Stufen

- Virtuelle Adress `0x7d_beef_cafe`: genau 39 Bit
- Binär:
  `0b0111_1101_1011_1110_1110_1111_1100_1010_1111_1110`

VPN[2]	VPN[1]	VPN[0]	Offset
502	503	252	

1. Lies `satp`-Reg. und finde Start der Seitentabelle auf Ebene 2 (PPN << 12)
2. Addiere `offset * size`,
   offset = VPN[2] = `0b1_1111_0110` = 502 * 8 = `satp` + 4016
3. Lies diesen Eintrag
4. Wenn `V(valid)=0`, dann erzeuge Seitenfehler
5. Wenn dessen `R`, `W`, oder `X`-Bits = 1, dann ist es ein Blatt, sonst Knoten
6. Die Adresse `PPN[2] | PPN[1] | PPN[0]` gibt an, wo die nächste Seitentabelle im physischen Speicher liegt
7. Wiederhole bei #2 bis ein Blatt gefunden wurde
8. Blatt: `PPN[2]`, `PPN[1]` und `PPN[0]` geben phys. Seitennummer an

Virtueller Speicher bei RISC-V

- Was geschieht, wenn eine Adresse nicht übersetzt werden kann?
  - Kein Eintrag für virtuelle Seitennummer vorhanden
  - Eintrag vorhanden, aber kein *valid* ("Gültig")-Bit
- Was geschieht, wenn ein PTE den Zugriff nicht erlaubt?
  - z.B. Versuch, in eine Seite zu schreiben, die nur das "R"-Bit gesetzt hat
- Ergebnis:
  Erzeugung einer Exception: Seitenfehler (page fault)
  - Kann beim Holen von Instruktionen sowie bei Laden und Speichern von Daten auftreten
  - Ein Instruktions-Seitenfehler tritt beim *instruction fetch* (→Pipeline) auf
    - Ungültiger Eintrag oder "X"-Bit nicht auf 1 gesetzt

Fazit zu virtuellem Speicher

- Virtueller Speicher ist nützlich für die **Isolation** zwischen Prozessen und **feingranulare Adressübersetzung**
  - Seitengranularität reduziert Übersetzungsaufwand
- Übersetzung durch MMU via Seitentabellen (+TLB-Cache)
  - Virtuelle Adressen von der CPU werden in physische Adressen übersetzt
  - Prozesse können identische virtuelle Adressen haben (z.B. beginnen alle Prozesse an der virtuellen Adresse 0, die dann auf separate Seitenrahmen für jeden Prozess abgebildet sind)
- Um Speicher zu sparen, sind Seitentabellen *dünn besetzt* und *hierarchisch*
  - RISC-V: zwei- (32 Bit) oder dreistufige (64-Bit) Adressübersetzung
    - Verschiedene Modi (sv39, 48, 57) verfügbar bei 64 Bit
  - Adresse der Wurzel der Seitentabelle im CSR `satp` gespeichert

Referenzen

[1] Edsger W. Dijkstra
Hierarchical ordering of sequential processes
Acta Informatica 1(2): 115–138, Juni 1971

[2] Palmer Dabbelt
Paging and the MMU in the RISC-V Linux Kernel
https://www.sifive.com/blog/all-aboard-part-9-paging-and-mmu-in-risc-v-linux-kernel

[3] Stephen Marz
The Adventures of OS: Memory Management Unit
https://osblog.stephenmarz.com/ch3.2.html