9. Virtueller Speicher

Michael Schöttner
Betriebssysteme und Systemprogrammierung

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

9.0 Vorschau

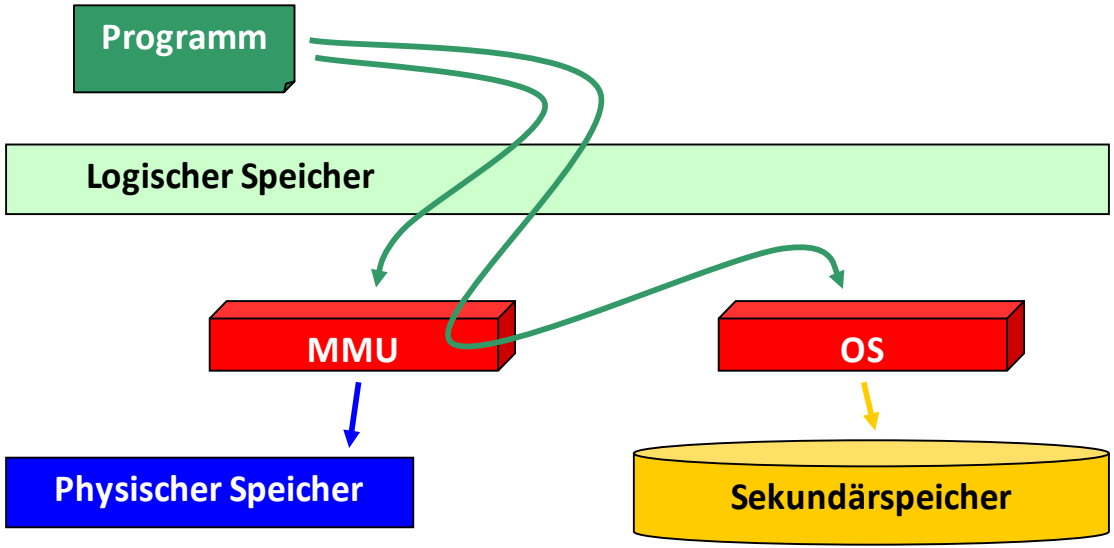
- Überblick
- Lokalitätsprinzip
- Paging
  - Seitentabellen und Adressübersetzung
  - Seitenauslagerungsstrategien
  - Kachelzuordnungsstrategien
  - Shared Memory
- Invertierte Seitentabellen
- Memory Management in Linux

9.1 Ziele

- **Virtualisierung**: Abstraktion von der Hardware, hier vom Hauptspeicher
- **Speichergröße**: mehr Speicher vorspiegeln, als physikalisch vorhanden ist
- **Speicherschutz**: für Prozesse untereinander und Betriebssystem
  - Prozess darf nur auf ihm zugeteilte Speicherbereiche zugreifen
  - Schutz vor unabsichtlichem und böswilligem Verhalten
- Mehrprogrammbetrieb: bedarfsabhängige dynamische Speicherzuteilung
  - physikalische Zuordnung von Arbeitsspeicher während Programmausführung anpassen
  - Ein- und Auslagerung von Teilen der Daten eines Prozesses
- Interne und externe Fragmentierung minimieren.

9.2 Komponenten

- Prozess arbeitet mit virtuellen/logischen Adressen
- CPU/MMU (CPU = Prozessor) und Betriebssystem übersetzen diese automatisch in physikalische Adressen



9.2 Komponenten

- Virtueller Speicher: Größe ist abhängig vom Prozessor
  - 32 Bit Prozessoren: 4 GB
  - 64 Bit Prozessoren: 256 TB (48 Bit, je nach Modell)
- Physikalischer Speicher: abhängig vom installierten Arbeitsspeicher
- Sekundärspeicher:
  - Dient dem Ein- und Auslagern von Speicherseiten (siehe später)
- MMU - Memory Management Unit (Funktionseinheit im Prozessor)
  - Erledigt Adressübersetzung
  - Überprüft ob Daten ausgelagert wurden
  - Und auch ob, unerlaubte Zugriffe stattfinden → Schutz

9.3 Adressraum

- = Menge fortlaufender Adressen die für den Prozessor zugreifbar sind

- Virtueller/logischer Adressraum:
  - In der Regel größer als der physikalische Adressraum
  - Aber die Daten sind unter Umständen gerade auf den Sekundärspeicher ausgelagert und müssen bei einem Zugriff erst von diesem geladen werden

- Physikalischer Adressraum:
  - Inhalte des virtuellen Adressraums sind hier gespeichert
  - Größe abhängig vom installierten Speicher

9.3 Adressraum

- Bisher: dynamische Partitionierung:
  - Programme können komplett ein- und ausgelagert werden
  - Die Anzahl der Partitionen bestimmt wie viele Programme maximal zu einem Zeitpunkt ausgeführt werden können

- Jetzt: virtueller Speicher
  - Teile eines Prozesses können ausgelagert werden
  - Fast keine Einschränkung hinsichtlich der Anzahl der gleichzeitig ausführbaren Prozesse
  - Prozesse, die „größer“ als der Arbeitsspeicher sind, können auch verarbeitet werden

9.4 Lokalitätsprinzip

- Bereits 1968 wurde das Datenzugriffsmuster von Programmen untersucht
- Zeitliche Lokalität:
  Zugegriffene Daten werden in naher Zukunft mit hoher Wahrscheinlichkeit wieder benutzt werden. → Caching sinnvoll
- Räumliche Lokalität:
  Nach Zugriff auf Speicheradresse `addr` ist ein Zugriff in der Nähe von `addr` wahrscheinlich → benachbarte Daten mit in den Cache laden

- Ursachen:
  - sequentielle Arbeitsweise von Programmen
  - Programme verwenden häufig Schleifen
  - Zugriff auf gruppierte Daten (z. B. Arrays)

9.5 Paging

- Logischer Adressraum wird in gleich große **Seiten (engl. pages)** unterteilt
- Physikalischer Adressraum wird ebenfalls in gleich große **Kacheln** (Seitenrahmen / engl. **page frames**) zerlegt
  - Die Daten einer Seite werden in einer Kachel gespeichert (sofern diese geladen ist)
  - Deswegen haben die Kacheln die gleiche Größe wie die Seiten
- Damit werden Prozesse in viele kleine Seiten unterteilt
  → große Flexibilität bzgl. Ein- und Auslagern

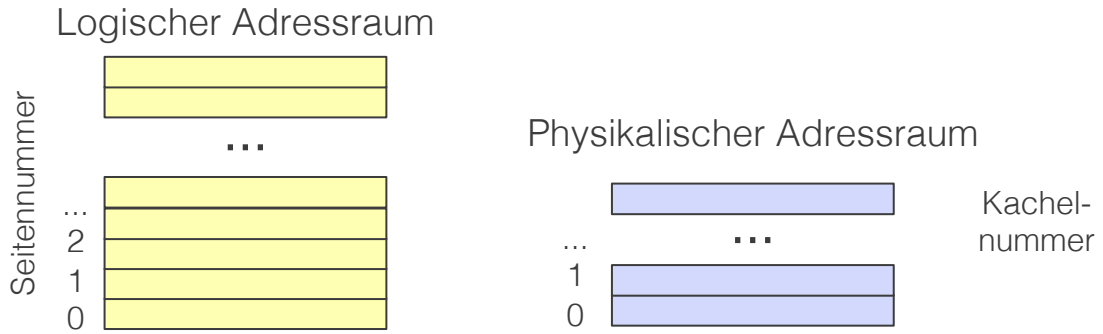
Weitere Entwurfsfragen

- Wie findet der Prozessor die richtige Kachel zu einer Seite?
- Wie funktioniert das Ein- und Auslagern?
- Wie wird entschieden was ausgelagert wird?

- Mehr dazu später

9.5.1 Adressübersetzung

- Eine 1:1 Abbildung, sprich Seite 0 → Kachel 0, Seite 1 → Kachel 1, ist nicht möglich, da der logische Adressraum (i.d.R.) größer als der physikalische Adressraum ist.



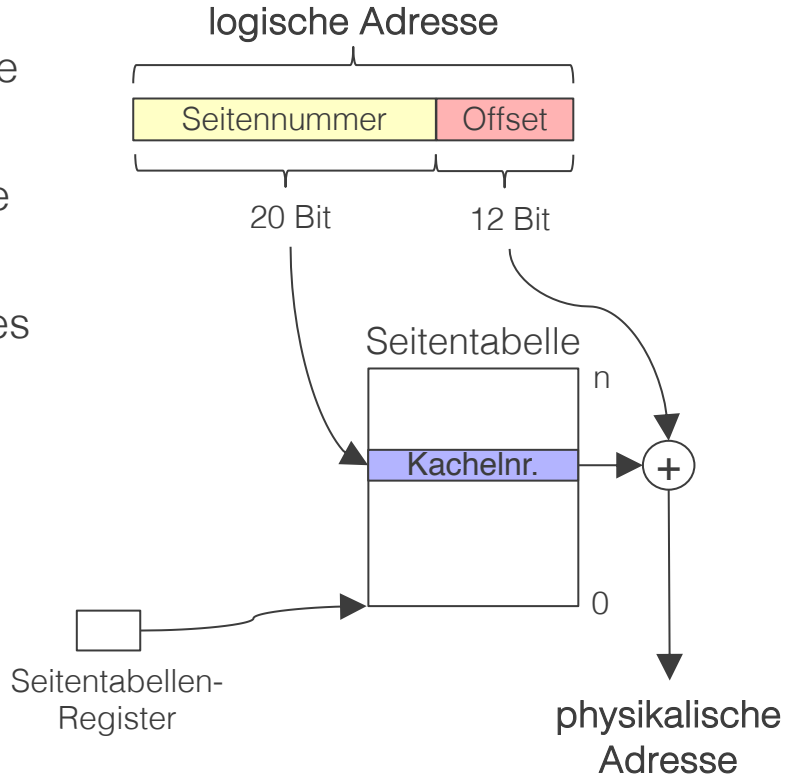
- Es ist somit eine Datenstruktur notwendig, um die Zuordnung von Seiten zu Kacheln zu speichern → **Seitentabelle**
  - Wichtig ist dabei, dass die CPU schnell „nachschlagen“ kann, welche Kachel zu welcher Seite gehört

Einfache Seitentabelle

- Eine Seitentabelle mit je einem Eintrag pro Seite
  - Hier steht die Kachelnummer
  - Sowie weitere Informationen, u.a. ob die Seiten geladen ist (mehr später)
- Ein Teil der logischen Adresse bildet die Seitennummer und dient als Index in die Seitentabelle → Adressübersetzung in O(1) möglich

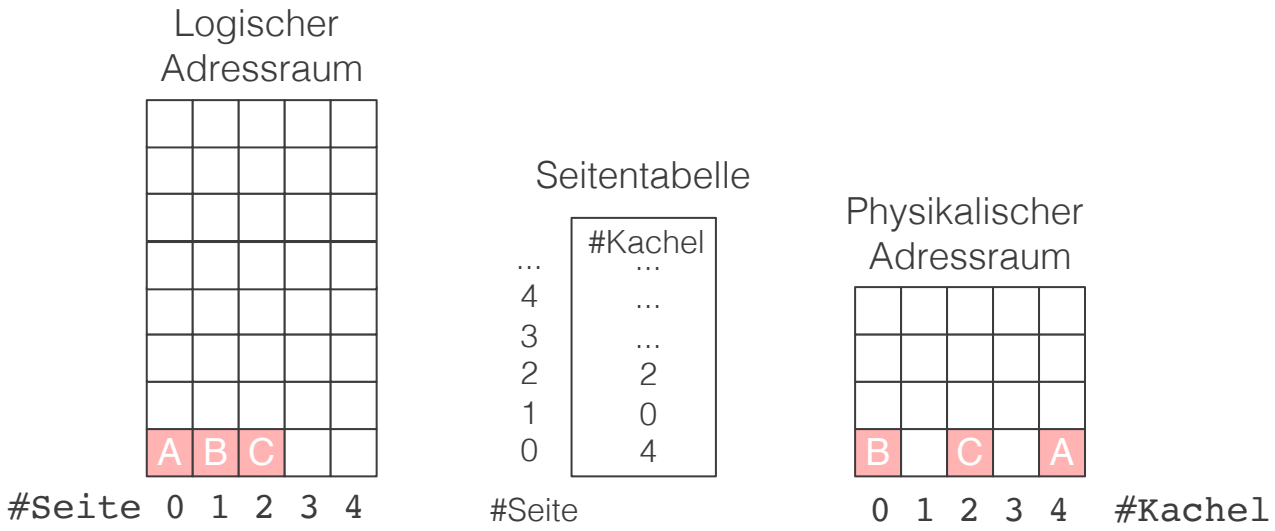
Einfache Adressübersetzung

- Beispiel: 32 Bit System, 4 KB Seiten
  - Offset bestimmt Byte innerhalb der Seite
    - 12 Bit erlauben Offset von 0 - 4095
  - Seitennummer dient als Index in Tabelle
    - Restlichen Bits: 32 – 12 = 20 Bits
  - Die Seitentabelle wird über ein spezielles Register der CPU gefunden



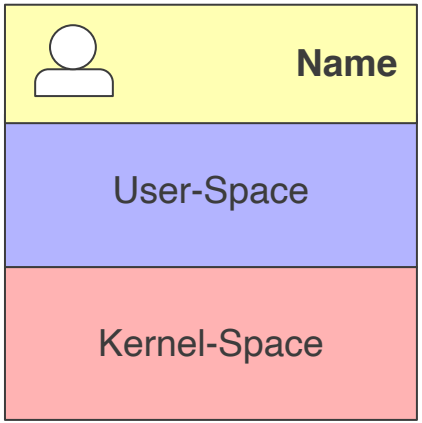
Einfache Adressübersetzung

- Aufeinanderfolgende Seiten müssen nicht unbedingt auf fortlaufende Kacheln abgebildet werden.
  - Damit entsteht keine externe Fragmentierung im physikalischen Speicher

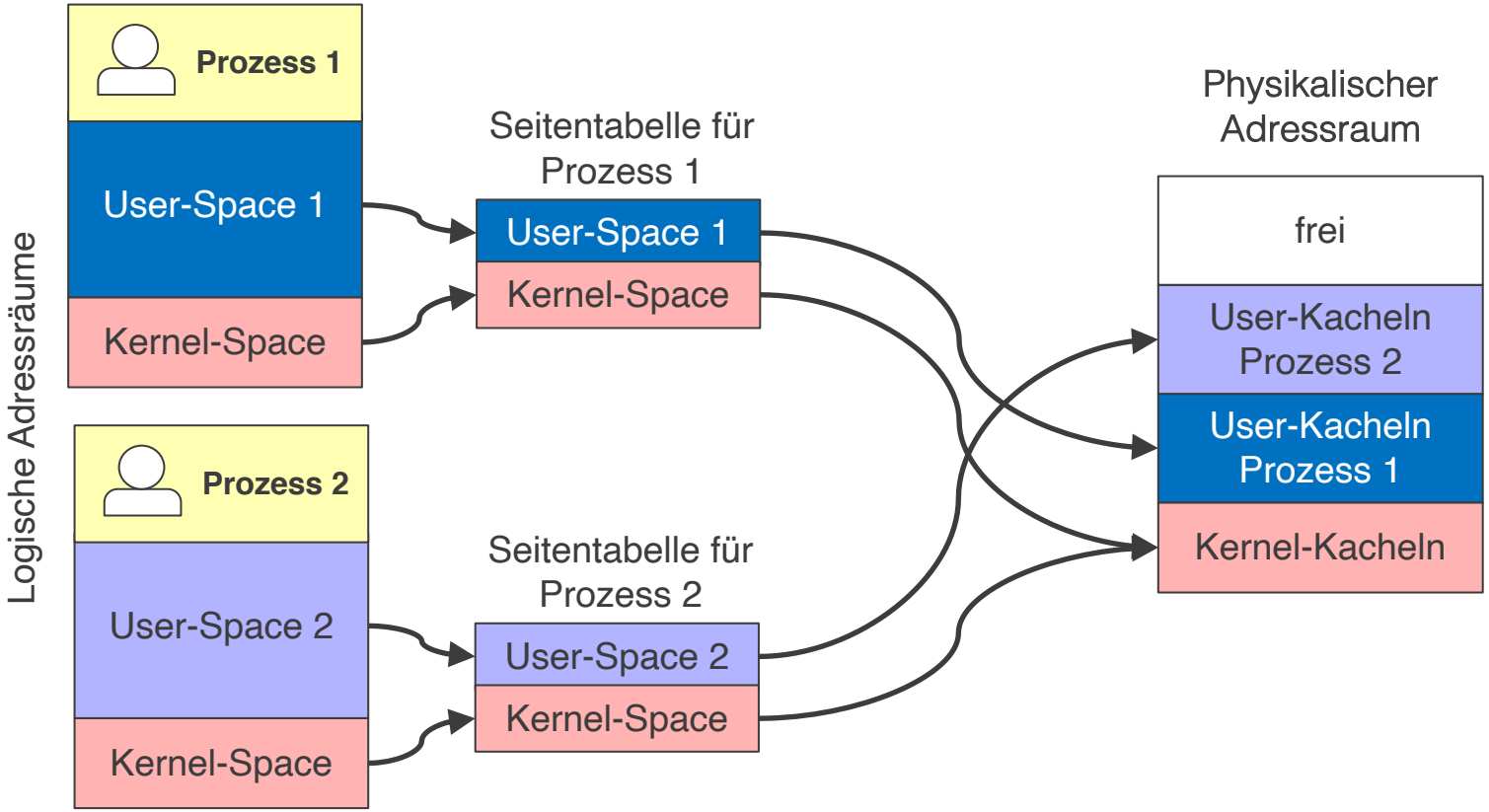


9.5.2 Prozesse

- Prozesse haben jeweils ihren eigenen logischen Adressraum
- Und benötigen deswegen jeweils eine eigene Seitentabelle
  - Hier steht die Zuordnung von Seiten zu Kacheln für den jeweiligen Prozess
  - Die Tabelle wird im Betriebssystem geschützt verwaltet
- Prozess-Darstellung aus Kapitel 5:

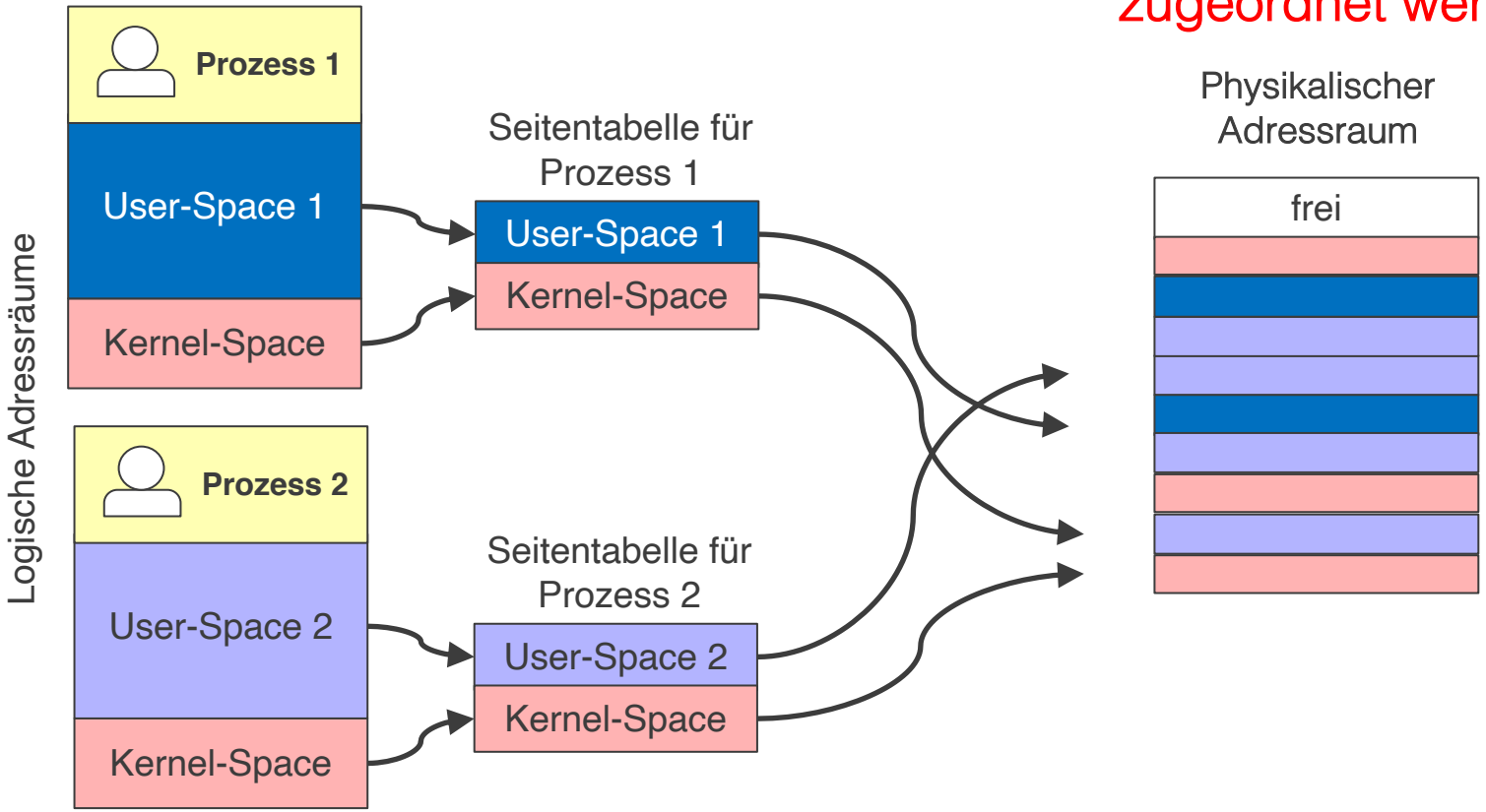


Mehrfach virtualisierter Speicher



Mehrfach virtualisierter Speicher

Kacheln können beliebig zugeordnet werden



Prozesse und einfache Adressübersetzung

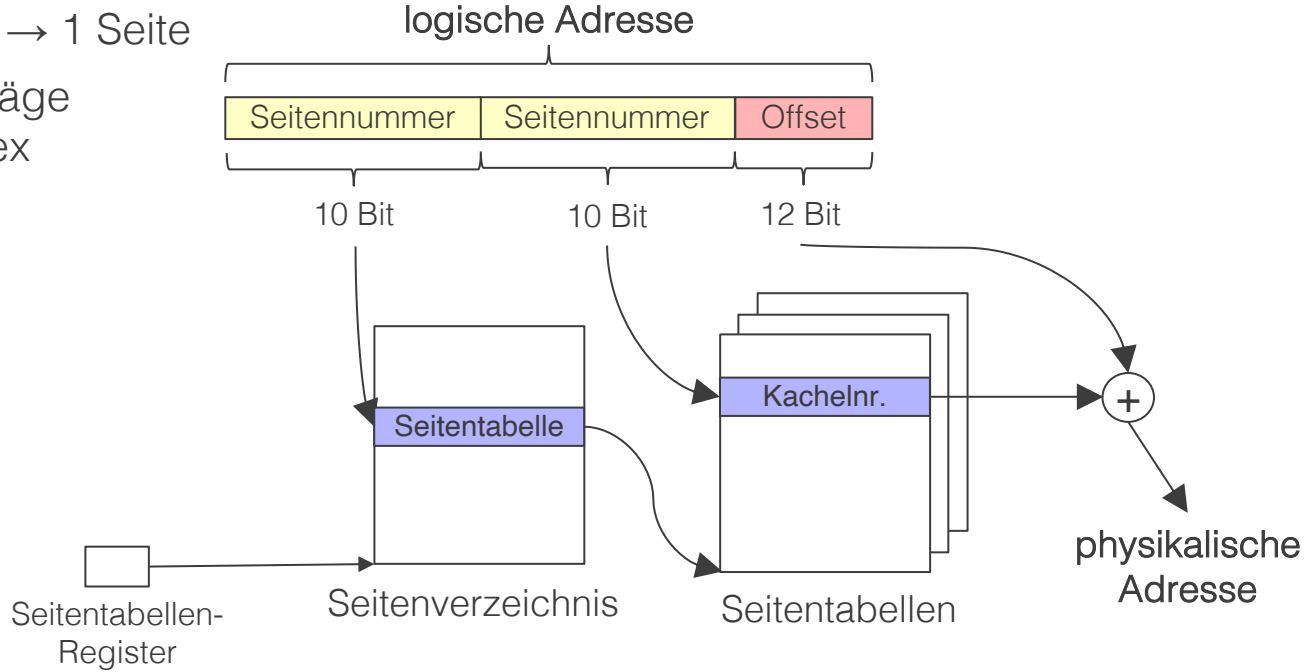
- Jeder Prozess benötigt seine eigene Adressübersetzungstabelle
- Dadurch entsteht ein großer Speicherverbrauch nur für die Tabellen
- Zahlenbeispiel:
  - 32 Bit System mit 4 KB Seiten
  - Virtueller Adressraum ist 4 GB groß
  - Unterteilt in 4 KB Seiten ergibt dies 1 Mio. Seiten
  - Damit hat eine Tabelle 1 Mio. Einträge, z.B. zu je 4 Byte
  - Damit benötigt eine Tabelle 4 MB physikalischen Speicher
  - Bei 100 Prozesse würden 400 MB RAM benötigt (bei 64 Bit noch viel mehr)
- Weitere Überlegung: kleine Prozesse benötigen nur einen Bruchteil ihres logischen Adressraums und trotzdem wird die komplette Tabelle angelegt

9.5.3 Zweistufige Adressübersetzung

- Lösung des Problems des großen Speicherverbrauchs durch hierarchische Seitentabellen.
- Ebene 1: Seitenverzeichnis (engl. page directory)
  - Jeweils ein Mal pro Prozess vorhanden
  - Diese Einträge zeigen auf je eine Seitentabelle der Ebene 2 oder sind leer (Hierbei werden physikalische Adressen verwendet)
- Ebene 2: Seitentabellen (engl. page tables)
  - Werden dynamisch bei Bedarf angelegt und gelöscht
  - Tabellen für den Kernel-Space sind immer präsent und werden prozessübergreifend gemeinsam genutzt

Beispiel: IA32 für x86

- 32 Bit, 4 KB Seiten, Eintrag in Seitentabelle und -verzeichnis hat 4 Byte
  - Seitenverzeichnis → 1 Seite
  - Seitentabelle → 1 Seite
  - Je 1024 Einträge
    → 10 Bit Index



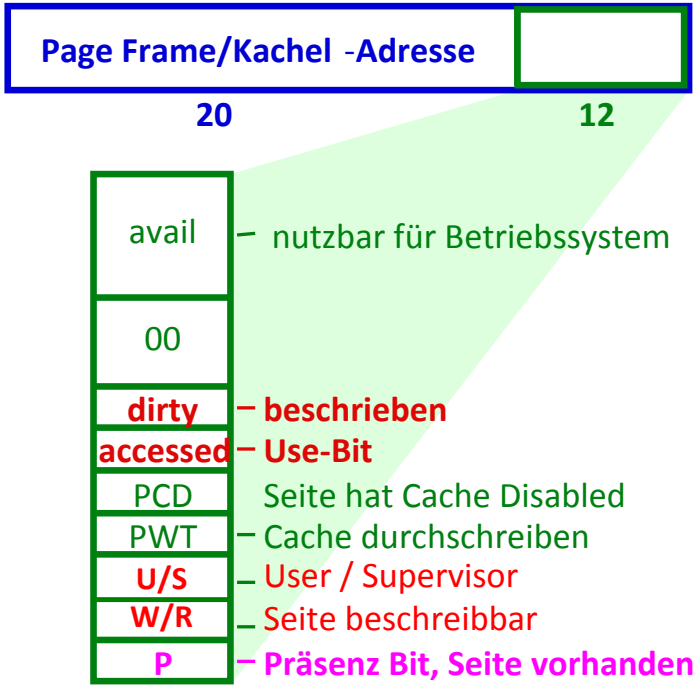
Bewertung

- Vorteil:
  - Speicherersparnis, da Seitentabellen nur bei Bedarf angelegt werden
- Nachteil:
  - Adressübersetzung ist jetzt langsamer, da nicht in einer, sondern zwei Tabellen nachgeschlagen werden muss

- Bei 64 Bit (IA32e) werden sogar 4 Stufen verwendet (siehe später)

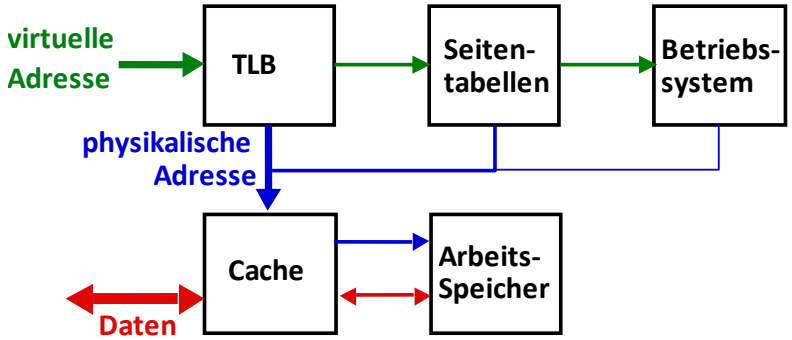
9.5.4 Format eines Seitentabelleneintrags bei IA32

- Kacheladresse nur gültig, wenn die Seite im Speicher vorhanden ist.
- dirty & accessed
  - Für Seitenersetzung (siehe später)
  - Werden von MMU gesetzt
- Präsenz-Bit
  - Zeigt an, ob Seite geladen ist
  - wird durch das BS verwaltet
- U/S-Bit
  - Zum Schutz des Kernels (siehe später)
  - wird durch BS verwaltet



9.5.5 Translation Lookaside Buffer (TLB)

- Ziel: Beschleunigen der Adressübersetzung
- Lösung: TLB → puffert früher übersetzte virtuelle Adressen
  - MMU sucht erst im TLB
  - Falls nichts gefunden, dann in den Seitentabellen
- TLB ist ein besonderer Schaltkreis welcher parallele Vergleiche realisiert → somit kann die MMU sehr schnell im TLB suchen
  - Hat nur wenig Einträge, z.B. 64 - 128



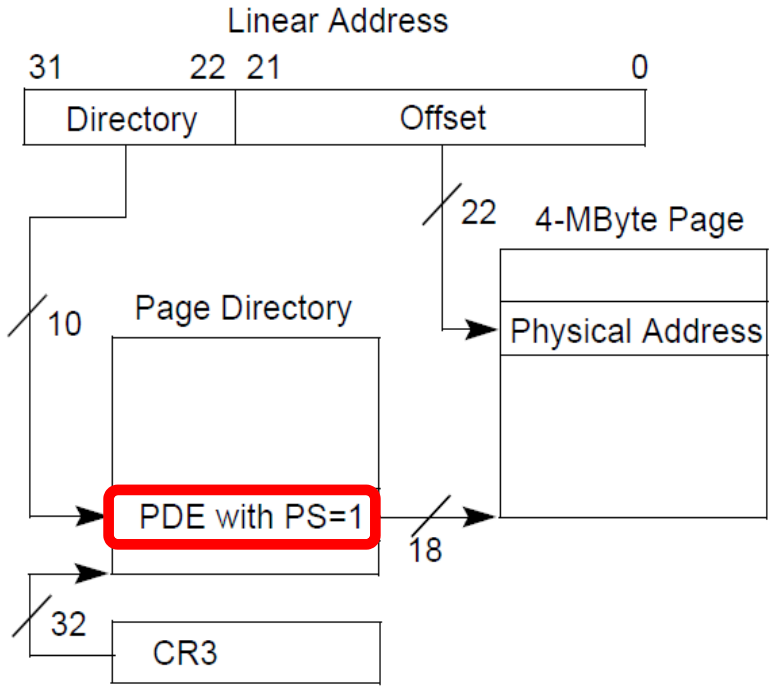
9.5.5 Translation Lookaside Buffer (TLB)

- TLB benützt nur physikalische Adressen:
  - unsichtbar für die Software
  - physikalischer Adresse ist eindeutig
    (evt. mehrere virtuelle Adressen für eine physikalische Adresse)

- TLB Programmierung:
  - komplett löschen bei Prozess-/Adressraumwechsel
  - einen Eintrag entfernen beim Auslagern einer Seite (siehe später)
  - Eintrag evt. fixieren, z.B. für Kernel (am besten 4 MB Seiten, siehe nächste Seite)
    - Adressübersetzung entfällt
    - Geht nur Teile die nie ausgelagert werden
    - Durch 4 MB-Seiten benötigt man nur wenig Einträge im TLB

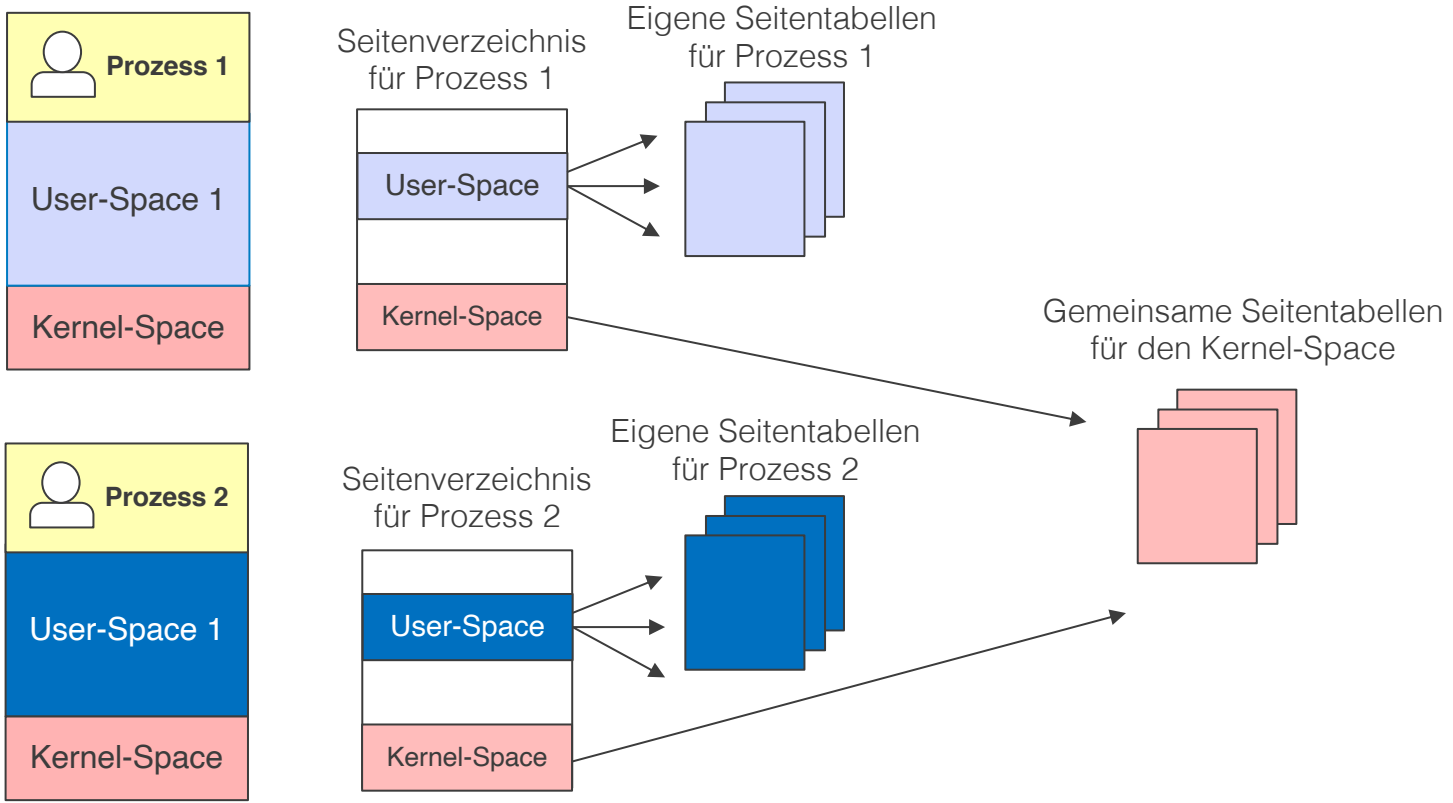
4 MB Seiten bei IA32

- Optional, muss extra aktiviert werden
- PS-Bit im in Einträgen des Page-Directory zeigt an, ob 4 KB oder 4 MB Seite



PDE = Page Directory Entry

9.5.6 Gemeinsame Tabellen für Kernel-Space



9.5.6 Gemeinsame Tabellen für Kernel-Space

- Pro Prozess wird ein Seitenverzeichnis benötigt → 4 KB
- Zusätzlich werden Seitentabellen separat pro Prozess nach Bedarf angelegt → eine Tabelle belegt 4 KB und kann 4 MB adressieren (1024 Kacheln)

- Seitentabellen für den Kernel-Space gibt es nur ein Mal und diese werden gemeinsam genutzt für alle Prozesse
  - Hier werden unter Umständen auch 4 MB Seiten verwendet, sodass hier weitere Tabellen entfallen

- Insgesamt ist der hierarchische Ansatz sehr speichereffizient

9.5.7 Getrennte Tabellen für Kernel-Space

- Nach Spectre und Meltdown wird der Kernel nicht mehr in jeden Prozess-Adressraum eingeblendet, sondern wird einem eigenen Adressraum realisiert
- Siehe später, Kapitel „Sicherheit“

9.5.8 Einlagerungsstrategien

- **Demand Paging**: Einlagerung bei Bedarf, also bei einem Seitenfehler
  - Nur die benötigten Seiten eines Prozesses werden eingelagert
- **Pre-Paging**: Einlagerung vorab
  - Versuch, hohe Seitenfehlerrate beim Lauf eines Prozesses zu vermeiden
  - Kosten/Nutzen-Verhältnis von Pre-Paging hängt davon ab, ob die in der Zukunft benötigten Seiten eingelagert werden können
  - Evt. bei einem Seitenfehler benachbarte Seiten auch einlagern → Lokalität
- Kombination von Pre- und Demand-Paging:
  - Pre-Paging zu Beginn der Programmausführung vermeidet anfängliche hohe Seitenfehlerrate für Programmcode, statische Daten, Teile des Heaps und Stacks
  - weitere Einlagerungen erfolgen durch Demand-Paging

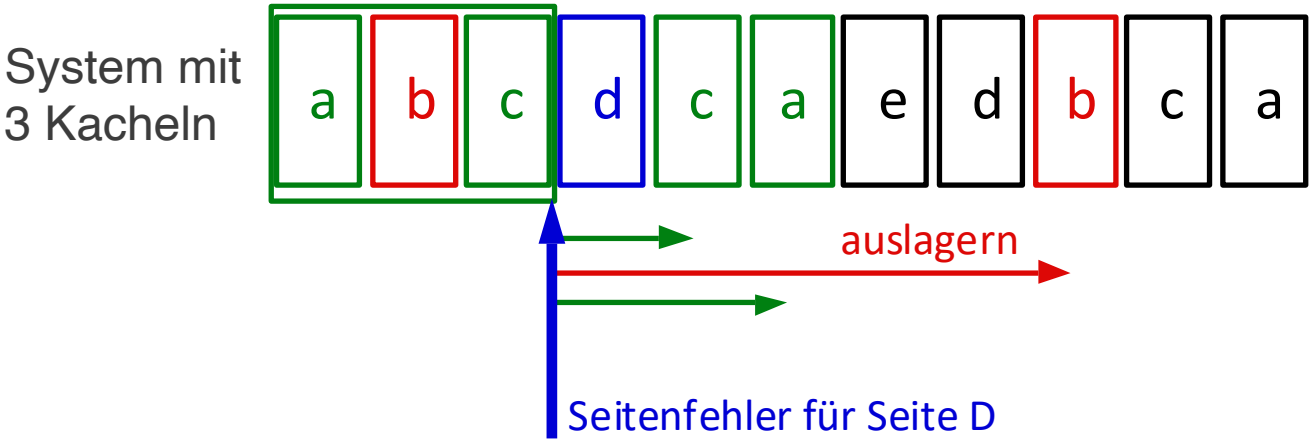
Ablauf eines Seitenfehlers (engl. page fault)

- Bei der Adressübersetzung erkennt die MMU, dass das Präsenz-Bit im Seitentabelleneintrag gelöscht ist und erzeugt eine Exception
  → **Seitenfehler (engl. page fault)**

- Dadurch wird ein Thread unterbrochen bevor der Speicherzugriff stattfindet und die Kontrolle an das Betriebssystem übergeben,
  an den **Page-Fault-Handler** (Funktion im Kern)
  - Der Page-Fault Handler lädt nun die fehlende Seite vom Sekundärspeicher, setzt das Präsenz-Bit und beendet die Ausnahmebehandlung
  - Dadurch wird das unterbrochene Programm wieder aktiviert und kann nun auf die Seite zugreifen

9.5.9 Seitenersetzung / Auslagern

- Falls eine Seite eingelagert werden muss (Seitenfehler), so muss evt. zuerst Platz geschaffen werden
- In diesem Fall muss eine andere Seite ausgelagert werden, aber welche?
- Ziel: Auslagern derjenigen Seite, die am längsten nicht mehr benötigt wird



9.5.9 Seitenersetzung / Auslagern

- Problem: Betriebssystem kann nicht in die Zukunft sehen
- Lösung: anhand des Speicherzugriffsverhaltens in der Vergangenheit wird versucht das Verhalten in der Zukunft abzuschätzen
- Überlegung: Seiten die in der Vergangenheit häufig zugegriffen wurden, werden vermutlich auch in der Zukunft noch benötigt

Strategie: Not recently used (NRU)

- Flags in der Seitentabelle:
  - **Accessed-Bit** falls die Seite referenziert wurde (periodisch zurücksetzen)
  - **Dirty-Bit** falls Seite verändert wurde (nicht periodisch zurücksetzen zeigt an, ob Seite auf Disk geschrieben werden muss beim Auslagern)
- Auslagerungspriorität nach Klassen:
  - A: Accessed-Bit = false, Dirty-Bit = false B: Accessed-Bit = false, Dirty-Bit = true
  - C: Accessed-Bit = true, Dirty-Bit = false D: Accessed-Bit = true, Dirty-Bit = true
- Beispiel: Priorität B beschreibt eine Seite, die in einem früheren Intervall verändert wurde und noch zurückgeschrieben werden muss.

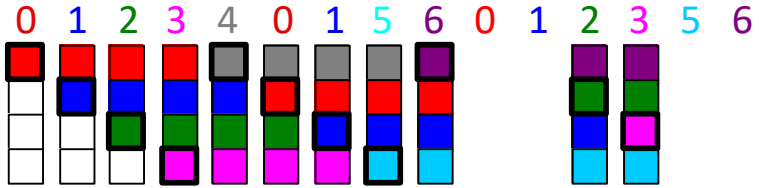
Strategie: First-in, First-out (FiFo)

- Idee: Die am längsten residente Seite wird ersetzt.
- Nachteile:
  - Auch häufig genutzte Seiten werden entfernt
  - Ungünstig bei zyklischen Zugriffsmustern
  - Evt. mehr Seitenfehler, falls mehr Kacheln zur Verfügung stehen (siehe nächste Seite)

Strategie: First-in, First-out (FiFo)

Belady's Anomalie (1969)

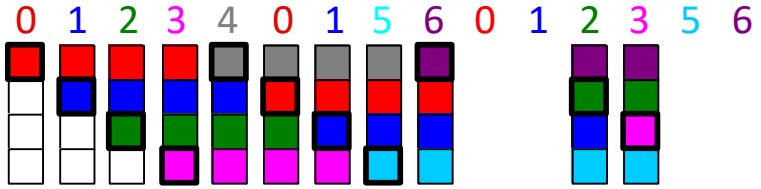
- 4 Kacheln → 11 Seitenfehler:



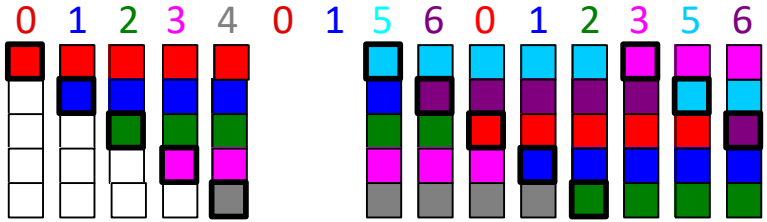
Strategie: First-in, First-out (FiFo)

Belady's Anomalie (1969)

- 4 Kacheln → 11 Seitenfehler:



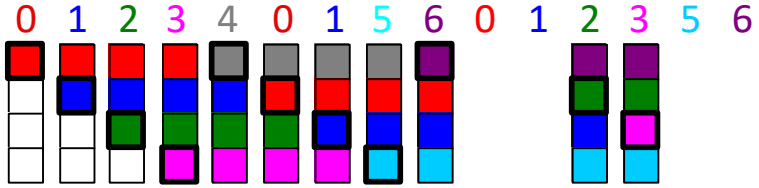
- 5 Kacheln → 13 Seitenfehler:



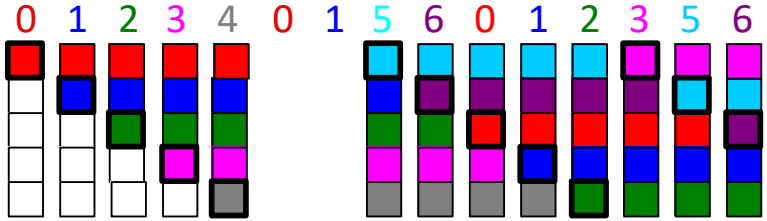
Strategie: First-in, First-out (FiFo)

Belady's Anomalie (1969)

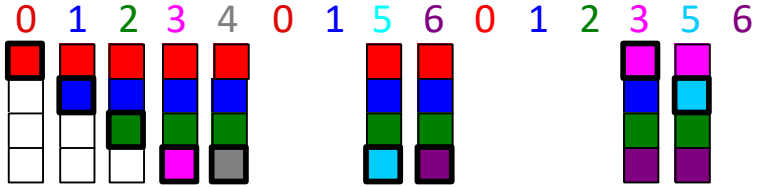
- 4 Kacheln → 11 Seitenfehler:



- 5 Kacheln → 13 Seitenfehler:

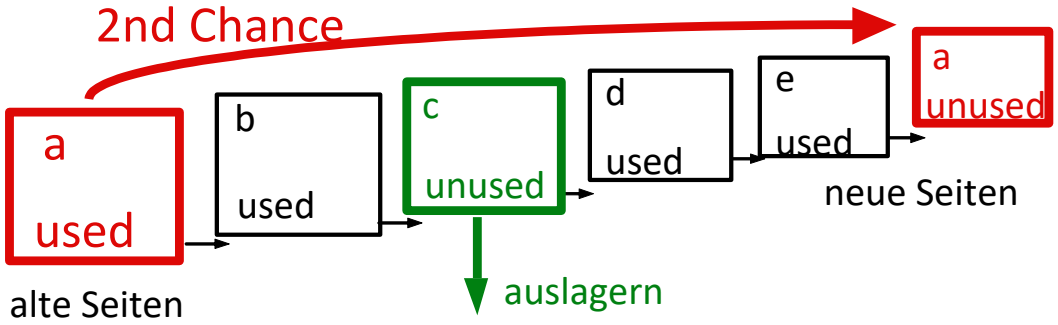


- Zum Vergleich optimale Strategie mit 4 Kacheln → 9 Seitenfehler:



Strategie: zweite Chance

- Verbesserung von reinem FiFo
- Second chance page replacement algorithm
  - Falls USE-Bit (= Accessed-Bit) gelöscht, dann Seite auslagern.
  - Falls USE-Bit gesetzt → **zweite Chance**
    - USE-Bit zurücksetzen und Seite hinten erneut einordnen



Strategie: zweite Chance

- Problem: Suche nach einem Auslagerungs-Kandidat dauert u.U. länger.

- Bemerkung:
  - Zurücksetzen der Used-Bits impliziert auch,
    dass betroffene Seiten aus TLB gelöscht werden
  - Andernfalls wird das Used-Bit beim nächsten Zugriff nicht gesetzt

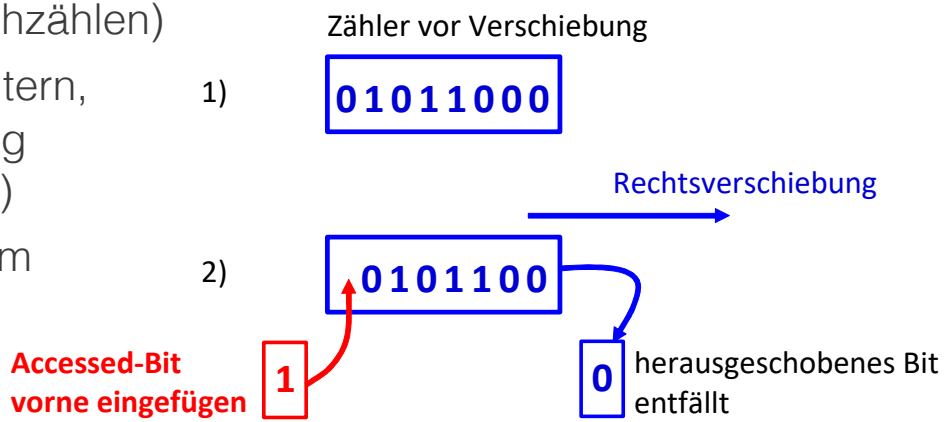
Strategie: Least Recently Used (LRU)

- Ziel: Am längsten unbenutzte Seite auslagern

- Erste Idee:
  - 64 Bit Zeitstempel in jedem Eintrag in der Seitentabelle
  - Hiermit den Zeitpunkt des letzten Zugriffs eintragen
  - Müsste bei jedem Zugriff aktualisiert werden → zu teuer
  - Außerdem würde dieser Ansatz einen beträchtlichen Speicheraufwand für die Zeitstempel verursachen

Strategie: Least Recently Used (LRU)

- LRU-Approximation in Software:
  - ursprünglich in Linux verwendet
  - pro verwendeter Seite ein 8-Bit Zähler
  - Accessed-Bit dient dem Hochzählen (Zugriff 1x pro Zeitscheibe, z.B. 50ms, in dem CPU erhalten wird, hochzählen)
  - Zusätzlich periodisch Zähler altern, durch eine Rechtsverschiebung (z.B. am Ende der Zeitscheibe)
  - Bei Bedarf Seite mit niedrigstem Zähler auslagern.



9.5.10 Seitenflattern (engl. thrashing)

- Falls ein Programm weniger Kacheln zur Verfügung hat, als es „ständig“ benötigt, so treten sehr oft Seitenfehler auf → Thrashing
- Mögliche Ursache: ein Programm hat zu wenig Kacheln

- Wie viele Kacheln soll ein Programm bekommen?

- Ziel: alle Programme sollten je nach Bedarf mehr oder weniger Kacheln bekommen und Beachtung der Fairness.

9.5.11 Zuordnung von Kacheln

- Jeder Prozess benötigt eine Mindestanzahl an Kacheln.
- Zuordnung: proportional, gleichverteilt, prioritätenabhängig, ...

- Lokale Strategie:
  - Prozesse haben eine feste Anzahl Kacheln zugeordnet
  - Bei einem Seitenfehler werden nur Seiten des betroffenen Prozesses ausgelagert
  - Vorteil: andere Programme werden nicht beeinträchtig
  - Nachteil: Kachelbedarf schwer schätzbar

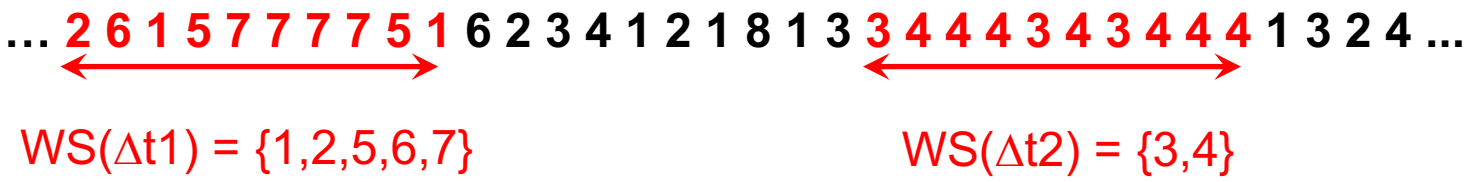
9.5.11 Zuordnung von Kacheln

- Globale Strategie:
  - Tritt ein Seitenfehler auf, so stehen die Seiten aller Prozesse zur Disposition.
  - Prozesse haben physikalischen Speicherbereich variabler Größe
  - Vorteil: mehr Flexibilität → Optimierung des Gesamtsystems
  - Nachteil: gegenseitige Beeinflussung des Paging-Verhaltens zw. Prozessen

- Ursachen von Thrashing
  - lokale Strategie: Zahl der Kacheln zu gering
  - globale Strategie: ein Prozess braucht zu einem Zeitpunkt sehr viele Kacheln, wodurch andere Prozesse beeinträchtigt werden

9.5.12 Working-Set-Modell

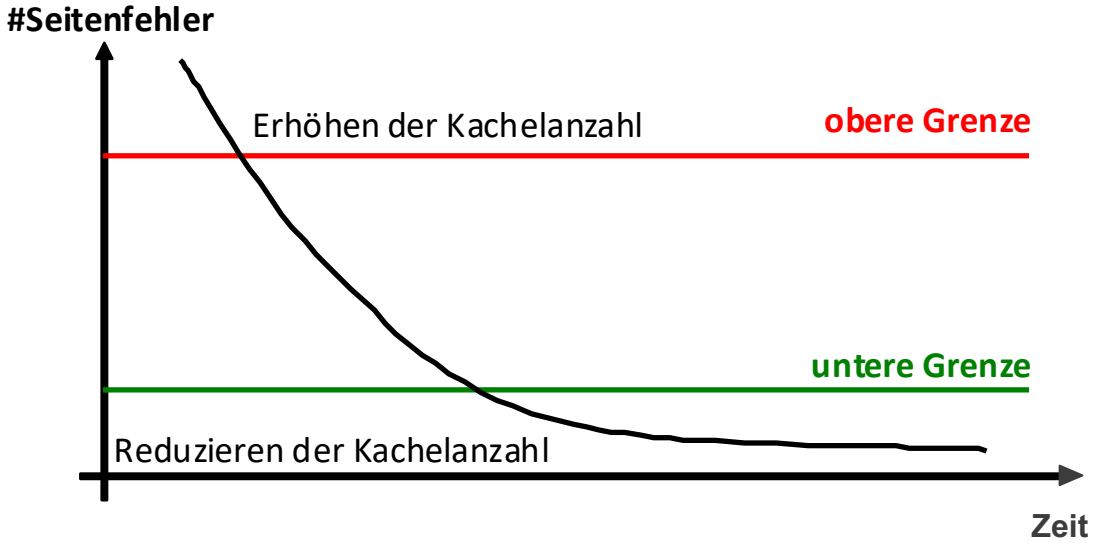
- = Arbeitsmengen-Modell → approximiert Lokalität.
- **Working Set (WS)** = Menge von Seiten, die in Δt referenziert werden.
- **Working Set Window (WSW)** = Menge zugeordneter Kacheln.
- Beispiel: Seitenzugriffssequenz & Working-Set



- Problem: Wahl des WSW → WS ändert sich ständig
  - zu klein: WSW umfasst nicht die gesamte Lokalität → Thrashing
  - zu groß: WSW umfasst mehrere Lokalitäten → weniger Prozesse effizient ausführbar

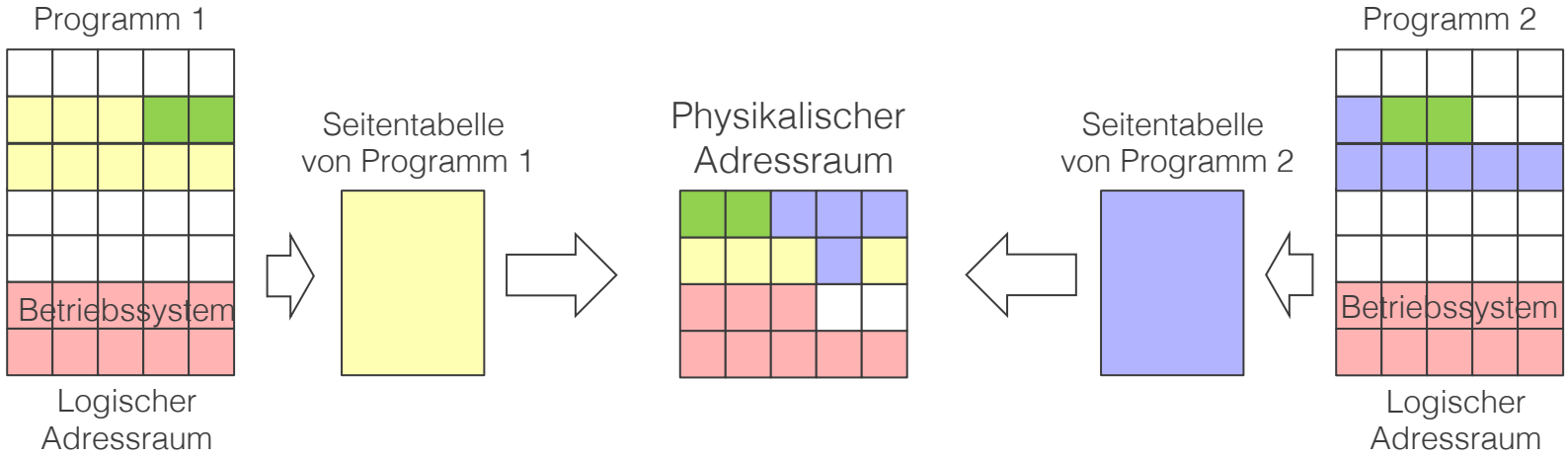
Page Fault Frequency Strategie

- Falls Seitenfehlerrate > oberer Schwellwert:
  - Allokation zusätzlicher Kacheln; wenn nicht mögl. Programm verdrängen
- Falls Seitenfehlerrate < unterer Schwellwert:
  - Freigabe von Kacheln
- Bem.: verwendet in Windows NT



9.5.13 Shared Memory

- Grundidee: eine oder mehrere logische Seiten verschiedener Prozesse nutzen die gleiche Kacheln
  - Die Einträge in den Seitentabellen der Prozesse verweisen auf die gleiche Kachel
  - Der Speicherbereich kann an der gleichen oder verschiedenen logischen Adressen eingeblendet werden



9.5.13 Shared Memory

- Achtung:
  - Zeiger innerhalb des Shared-Memory-Bereichs sind zulässig, sofern der Bereich in jedem Prozess an der selben logischen Adresse eingeblendet wird
    - Nützlich, um dynamische Datenstrukturen direkt im Shared-Memory abzulegen
  - Zeiger im Shared-Memory-Bereich dürfen niemals auf logische Adressen außerhalb des Shared-Memory verweisen, da diese nicht in jedem Adressraum gültig sind

- Shared-Memory muss beim Auslagern berücksichtigt werden
  → betrifft mehrere Prozesse (auch bei einer lokalen Kachelzuordnung)

Copy-On-Write

- Ziel: Speicher nur zum Lesen gemeinsam nutzen.
  - Schreibt ein Prozess, so wird die zugehörige Seite kopiert
  - Änderungen für andere Prozesse unsichtbar.

- Wichtig u.a. für `fork`:
  - Statt alle Daten zu kopieren, wird nur die Seitentabelle kopiert
  - Zusätzlich werden beim Eltern- und Kind-Prozess alle Seitentabelleneinträge für den User-Space auf read-only gesetzt.
  - Schreibt nun der Eltern- oder Kind-Prozess so erfolgt ein Protection-Fault (siehe später), das Betriebssystem wird aktiviert und erstellt eine Kopie der Seite und lässt dann den Zugriff zu. Damit sind die Adressräume getrennt und die Daten werden nur bei Bedarf umkopiert (deutlich schneller)

9.6 Invertierte Seitentabellen

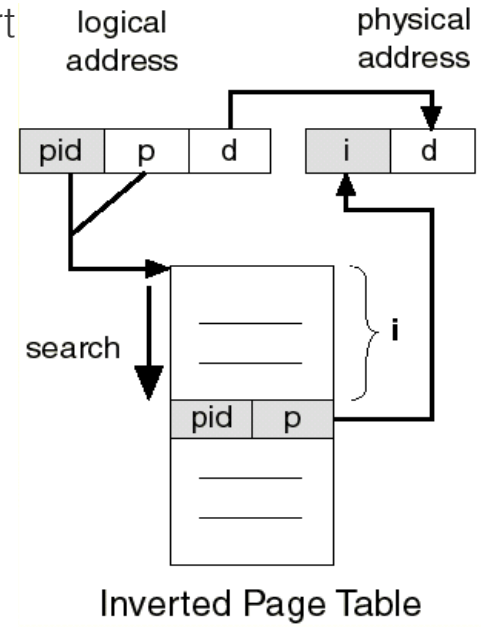
- Rechenbeispiel für vollständige Seitentabellen für 64-Bit
  - ~ 2^{52} Einträge in Seitentabelle (12-Bits für Offset in 4 KB Seite)
  - ~ für 4 KB Seiten und 8 Byte pro Eintrag
  - ~ 32.000 Terabyte nur für Seitentabellen

- Besser einen Eintrag pro Kachel als Eintrag pro logischer Seite in einer Tabelle anlegen (Hauptspeicher ist kleiner als logischer Adressraum)

- Bemerkung zum Rechenbeispiel:
  - i.d.R. wird nicht der gesamte logische Adressraum benötigt
  - Und Seitentabellen ab Ebene 2 dürfen ausgelagert werden

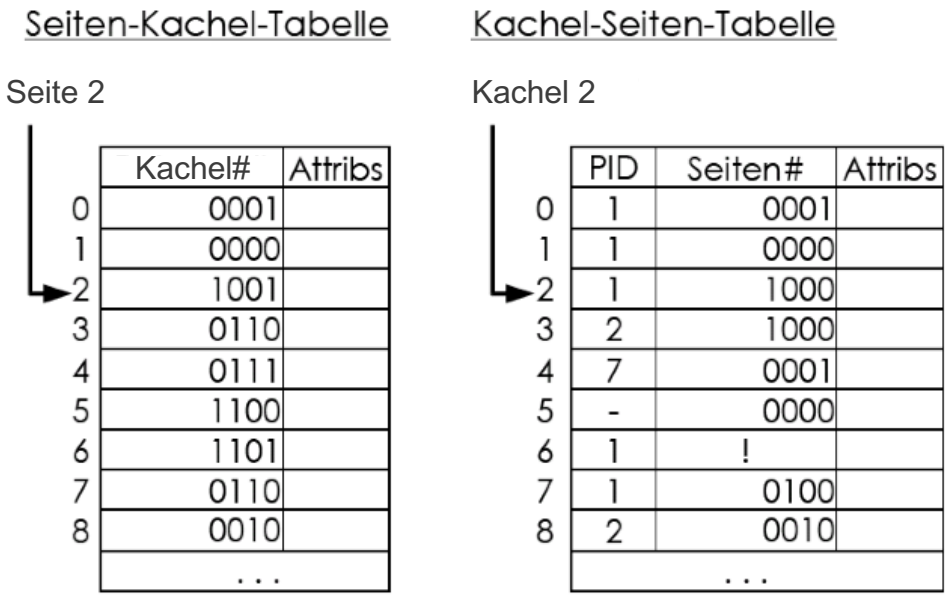
9.6 Invertierte Seitentabellen

- Invertierte Seitentabelle mit einem Eintrag pro physikalischer Kachel
  - 1. Eintrag ist Kachel 0, 2. Eintrag ist Kachel 1, usw.
  - Pro Eintrag ist die zugehörige logische Seite gespeichert
  - i.d.R. nur eine Tabelle für alle Prozesse
    → PID mit jedem Eintrag abspeichern
  - Im Bild: lineare Suche in Einträgen (langsam)
  - Schneller mit vorgeschaltetem TLB



9.6 Invertierte Seitentabellen

- Vergleich: Seitentabelle vs. invertierte Seitentabelle
  - Bild: Prof. Herrmann Härtig (TU Dresden)



Beispiel: IA64 (Itanium) != Intel 64 Bit Architecture

- MMU sucht zunächst in TLB (z.B. 128 Einträge für je Code & Daten)
  - TLB wird in Software verwaltet (nicht automatisch über die MMU)
- Falls kein Eintrag im TLB gefunden VHPT durchsuchen:
  - VHPT = Virtual Hash Page Table
  - Prozessor greift nur lesend zu
  - Aktualisieren und Konsistenz zw. VHPT und TLB ist Aufgabe des Betriebssystems
  - Tabelle liegt im virtuellen Adressraum → evt. TLB Miss Fault beim Durchsuchen.
- Bei einem TLB Miss Fault wird Betriebssystem aufgerufen
  - Muss dann passende Seite einlagern & Hashtabelle aktualisieren
  - Wie das im BS realisiert wird, ist offengelassen, z.B. Baumstruktur oder wieder hierarchische Tabellen

9.7 Linux 2.6 (32 Bit)

Physikalische Speicherverwaltung

- Physikalischer Speicher ist in drei Zonen unterteilt (ausgelassen)
- Pro Zone wird jeweils eine Buddy-Verwaltung eingesetzt

Buddy-Verwaltung

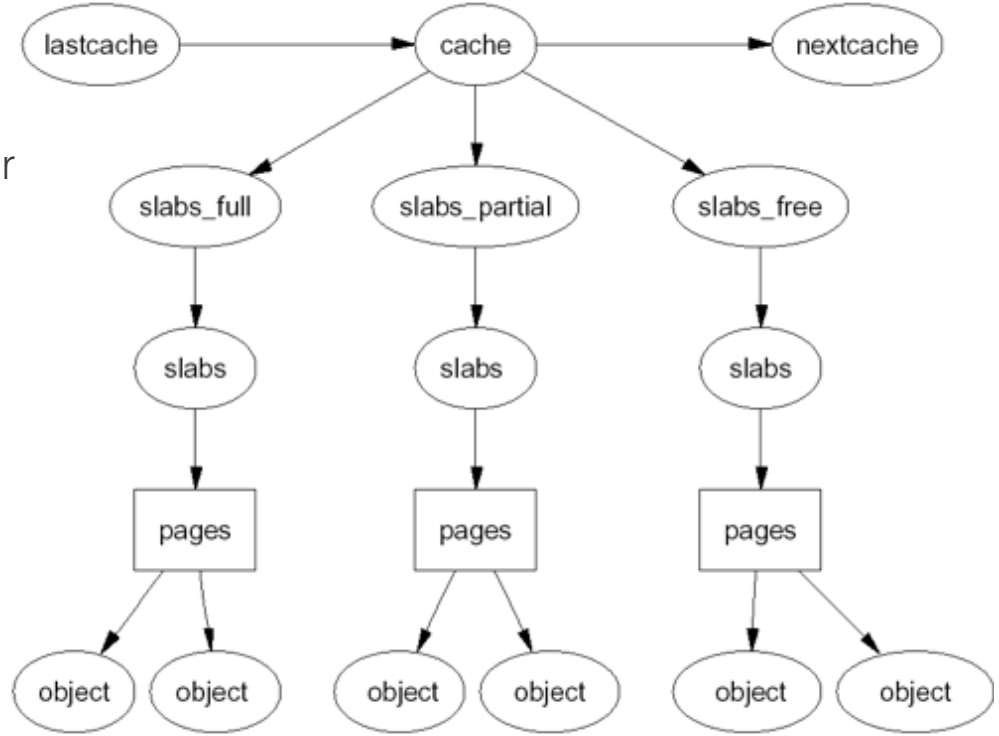
- 2^n-Byte großen Container; n={12, …, 23} → 4 KB – 8 MB
- Leere Blöcke schnell aggregierbar
- Freie Blöcke schnell auffindbar
- Alloziert Kacheln fortlaufend

Slab Allocator

- Gruppiert vorinitialisierte Objekte gleichen Typs → gleiche Größe, z.B. I-Nodes
- Pro ‚slab' eine oder mehrere Kacheln vorsehen
- Schnelles Auffinden von Blöcken mit passender Größe
- Vorteilhaft zur Allokation von kleinen Objekten
- Speicherallokation in den Größen 2^x (x>5)
- Mildert interne Fragmentierung
- Übernommen von Solaris (1994)

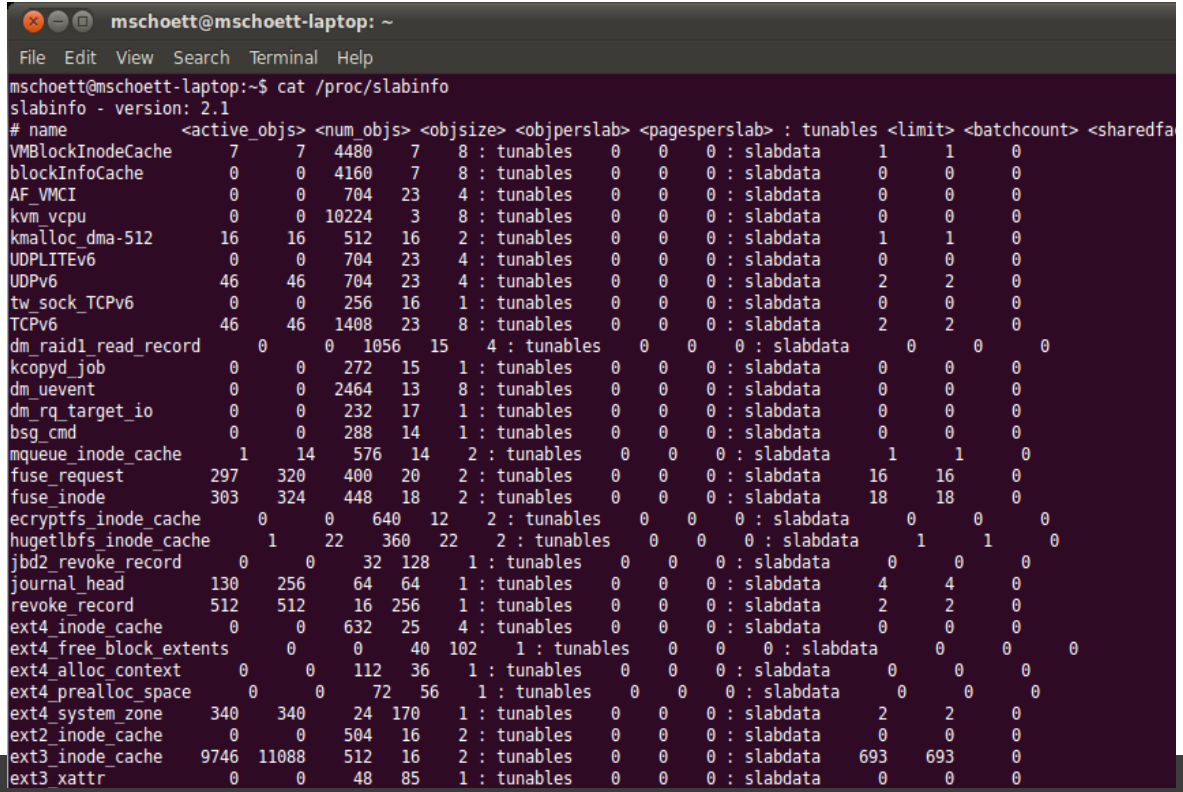
Slab Allocator

- "Caches" im Sinne von Behälter:
  - für einen Objekttyp/-größe
  - mehrere ‚slabs' pro Cache
  - Slabs: voll, partiell gefüllt, leer



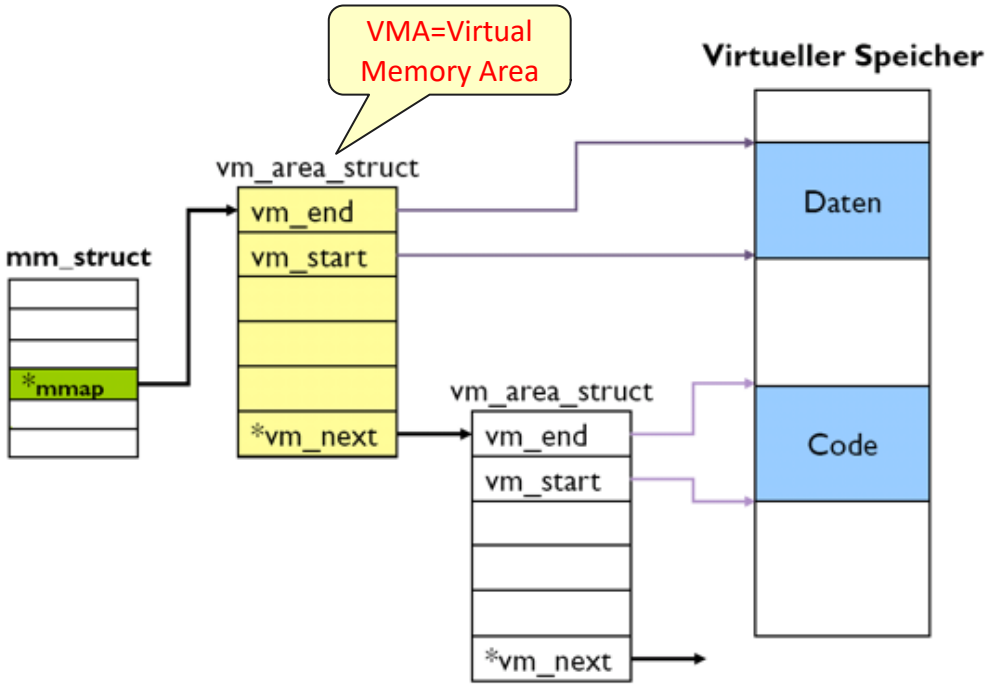
Slab Allocator

- Ausgabe von `cat /proc/slabinfo`



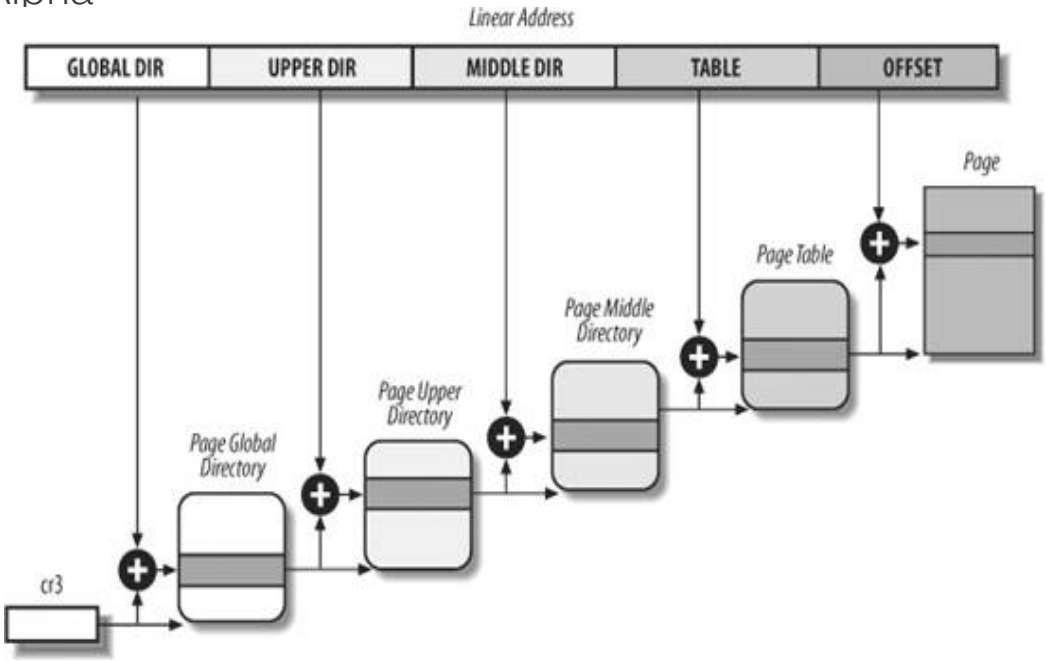
Virtuelle Speicherverwaltung

- Unterteilung des virtuellen Adressraums (32 Bit):
  - User Mode: 0-3GB (privat pro Prozess), Kernel Mode: 3-4 GB (shared)
  - Schutz des Kerns über das Supervisor-Bit in Seitentabelleneinträgen
- Adressraumverwaltung → eines Prozesses
  - Pro VMA Berechtigungen r, w, x
- Anzeigbar mit: `cat /proc/<pid>/maps`



Paging: Demand und Pre-Paging

- Ursprünglich dreistufige Seitentabellenstruktur
  - entwickelt für 64-Bit Alpha
- Ab Kernel 2.6.11 vierstufige Tabellen (für IA32e 64-Bit Mode)



Seitenersetzung

Vier Typen von Seiten:

- **Unreclaimable** Seiten nicht auslagerbar → locked pages, kernel stacks …
- **Swapable** (normal data) Seiten müssen vor dem Auslagern in Swapdatei zurückgeschrieben werden
- **Syncable** (memory-mapped files) Seiten in Datei zurückschreiben, falls als „dirty“ markiert
- **Discardable** Seiten sind unbenutzte Seiten, können direkt verworfen werden

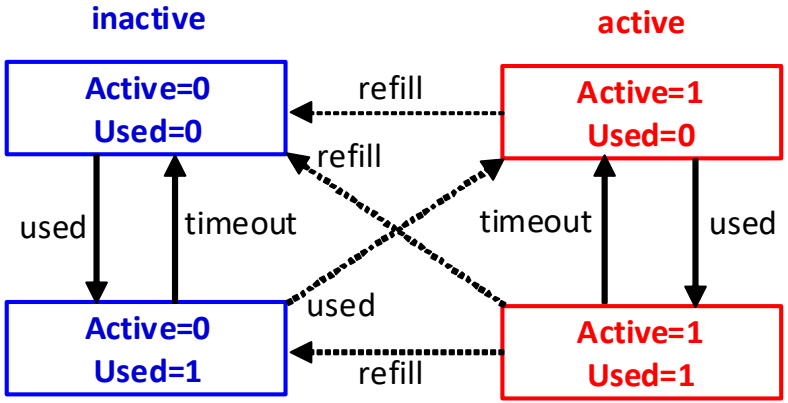
Seitenersetzung

- 1. Periodic reclaiming (kswapd)
  - Kernel thread
  - prüft alle 10s Watermarks der Kachel-Zonen → bei Bedarf PFRA starten
- 2. Low on memory reclaiming
  - Low-Memory-Situation wird durch Kernel erkannt → sofort PFRA ausführen

- Page Frame Reclamation Algorithm (PFRA)
  - versucht zunächst discardable oder „inactive“ pages auszulagern (nächste Folie)
  - bei Bedarf aber auch swapable und syncable pages
  - Ziel: pro Durchlauf 32 Kacheln freimachen

Seitenersetzung

- PFRA verwendet zwei Listen: active_list und inactive_list pro ZONE
  - Used-Bit: wird periodisch für alle Einträge zurückgesetzt
  - Active-Bit: gibt Listenzugehörigkeit an; Übergang gesteuert durch PFRA → refill
- active_list:
  - häufiger verwendete Seiten
  - Working-Sets aller Prozesse
  - Verschiebung nach inactive_list, wenn Used=0 und PFRA erneut prüft
- inactive_list:
  - seltener verwendete Seiten
  - gute Kandidaten für eine Auslagerung
  - Verschiebung nach active_list, wenn Used=1 und erneuter Zugriff stattfindet



Bemerkungen zur Seitenersetzung

- PFRA kann notfalls aus allen Zuständen auslagern
- Code-Kacheln des Kern-Images werden aber nie ausgelagert
- Slabs werden gesondert behandelt

- Pdflush ist ein weiterer Dämon der Speicherverwaltung
  - Läuft ca. alle 500ms
  - Schreibt vorsorglich Dirty-Pages zurück auf Disk
    → damit reduziert sich der Aufwand beim Auslagern

Weiterführende Informationen

- Linux Memory Management (Webseite), http://linux-mm.org
- Buch: „Understanding the Linux® Virtual Memory Manager“, Mel Gorman, Prentice Hall, 2004; frei als PDF verfügbar

9.8 Hörsaal-Übung

- Wir möchten mithilfe eines Kernel-Moduls Informationen über den Speicher eines Prozesses abrufen und ausgeben.

- Hierzu sind einige Hintergrundinformationen notwendig, zu Kernel-Modulen und auch zu den relevanten Datenstrukturen, die uns interessieren

Task-Management: `struct task_struct`

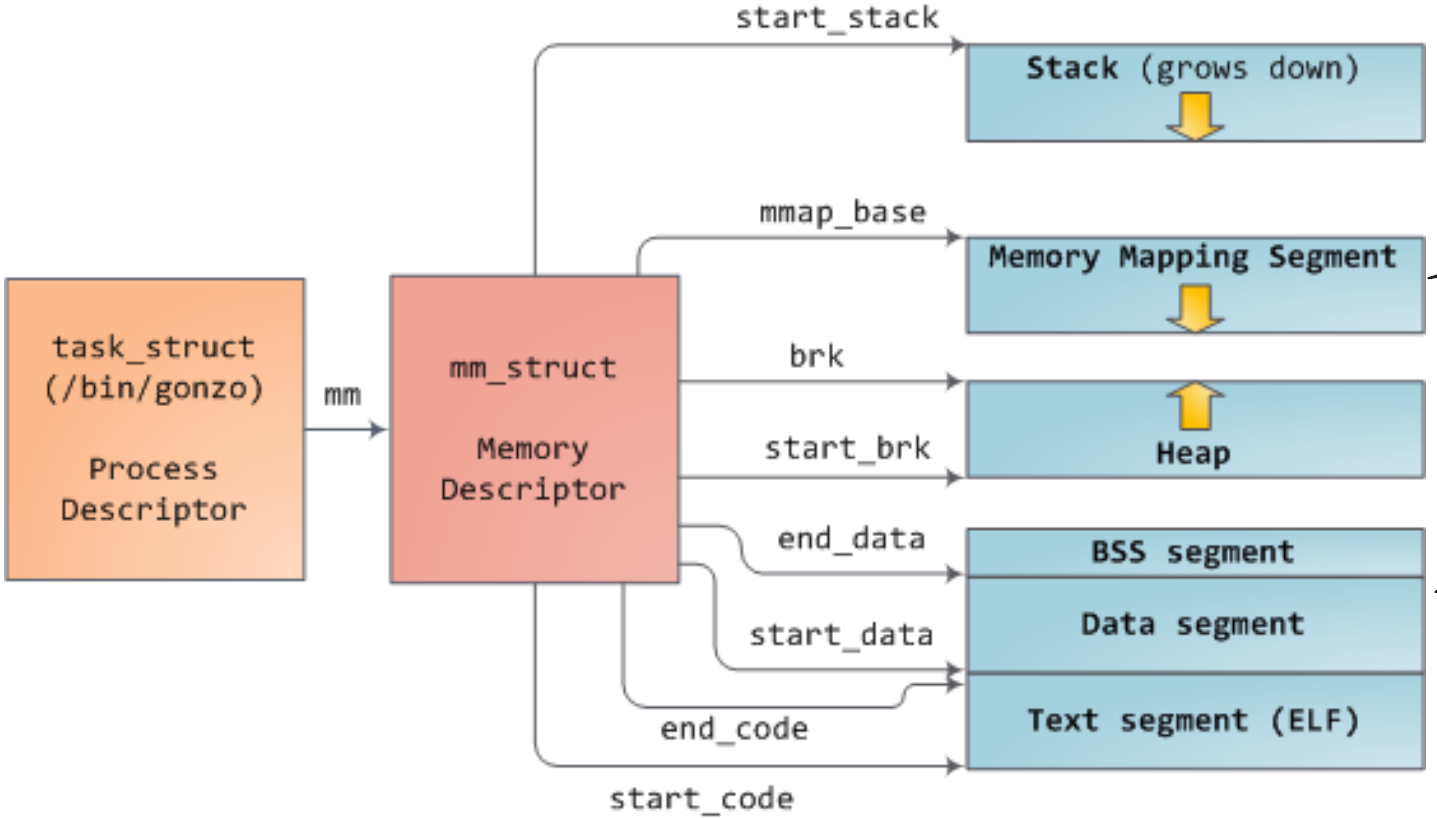
- Haupt-Struktur für einen Prozess/Thread
  - Wenn Prozess startet gilt pid = tgid
  - Für alle anderen Threads gilt pid = thread id -> tgid ist leader

```
#include <include/linux/sched.h.h>

struct task_struct {
   volatile long state;    /* runnable, stopped, ... */
   ...
   pid_t pid;              /* processThread id       */
   pid_t tgid;             /* thread group id        */
   ...
   char comm[16];          /* name of process        */
   ...
   unsigned long policy;   /* scheduling policy      */

   struct mm_struct *mm, *active_mm; ;  /* memory descriptor */
```

Überblick der Speicherstrukturen eines Prozesses



Bildquelle: https://manybutfinite.com/post/how-the-kernel-manages-your-memory/

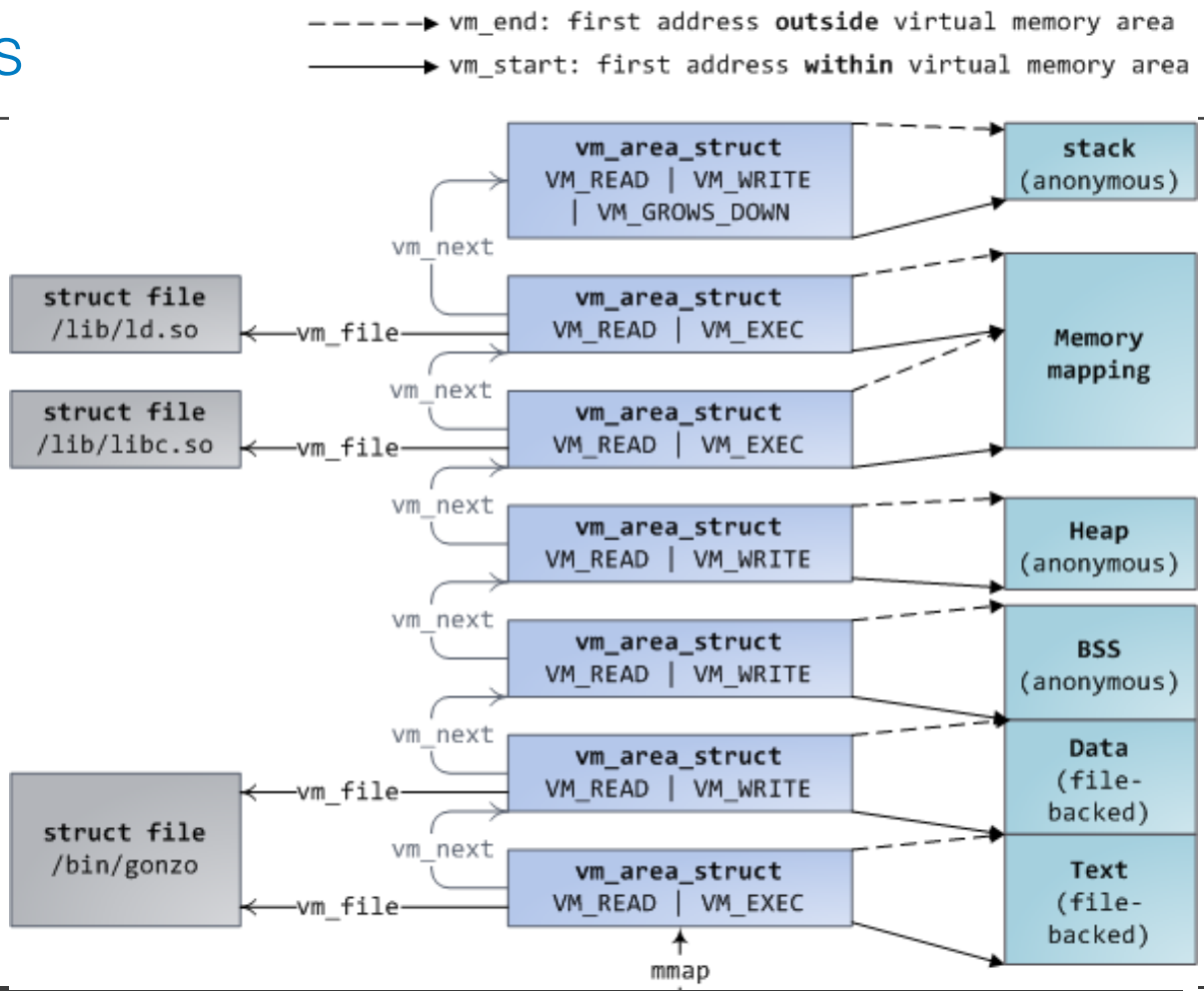
Memory-Management: `struct mm_struct`

- Speichersegmente, Einstieg in VMA-Liste, Zeiger auf Page-Directory

```
#include <include/linux/sched.h.h>

struct mm_struct {
   struct vm_area_struct *mmap;          /* list of VMAs */
   ...
   unsigned long start_code, end_code,
                 start_data, end_data;   /* globals */
   unsigned long start_brk, brk,         /* heap -> see brk(addr)
                                            to increase heap size */
                 start_stack;
   pgd_t *pgd;                           /* pointer to page directory */
   ...
};
```

Beispiel: VMAs



Bildquelle: https://manybutfinite.com/post/how-the-kernel-manages-your-memory/

Virtual Memory Area: struct vma_struct

- Bereich im virtuellen Adressraum
  - Zwei VMAs überlappen nicht
  - Flags: Berechtigungen: VM_READ, VM_WRITE, VM_EXEC und VM_SHARED

```
#include <include/linux/mm.h>

struct vm_area_struct {
   struct mm_struct *vm_mm;      /* The address space we belong to. */
   unsigned long vm_start;       /* Our start address within vm_mm. */
   unsigned long vm_end;         /* The first byte after our end address
                                    within vm_mm. */
   ...
   /* linked list of VM areas per task, sorted by address */
   struct vm_area_struct *vm_next;

   struct file *vm_file;        /* reference to file if any */
   ...
}
```

Installieren und Deinstallieren von Kernel-Modulen

- Module können dynamisch zur Laufzeit in den Kern geladen werden
  - Superuser-Rechte sind hierfür notwendig
- Modul dynamisch laden mit: `insmod myModule.ko`
- Modul entladen mit dem Befehl: `rmmod myModule`
- Alle geladenen Module kann jeder Benutzer mit `lsmod` auflisten.

Initialisierung und Beenden von Modulen

- Unterschiede zu herkömmlichen Usermode-Programmen:
  - Keine `main`-Funktion vorhanden.
  - Funktionen der Standard C Bibliotheken nicht verfügbar.

- **Init-Routine**: definiert durch Makro `module_init(myInit);`
- **Cleanup-Routine**: definiert durch Makro `module_exit(myExit);`
- Verschiedene optional Makros vorhanden:
  - z.B. Angabe der Lizenz MODULE_LICENSE(“GPL”) …

Debug-Ausgaben

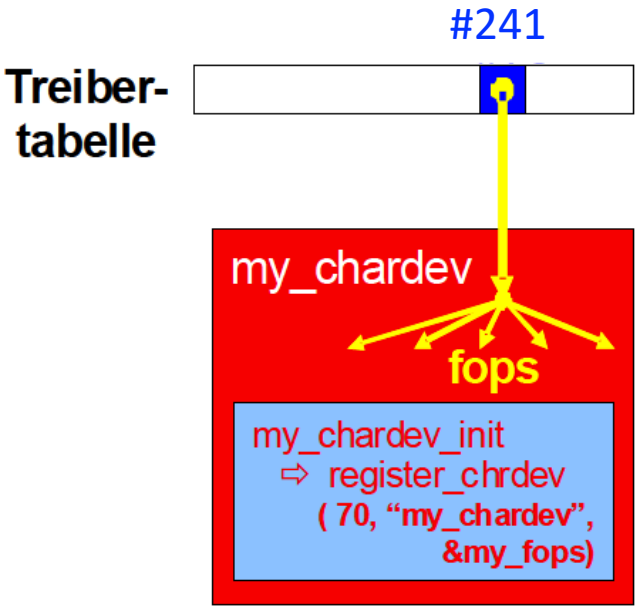
- Ausgaben mit **`printk`** statt mit **`printf`**
- Priorität der Nachricht kann in der Zeichenkette definiert werden
  - Bsp.: **`printk(''<0> panic \n''); // <0> = kernel emergency`**
- Nachrichten werden im Kernel-Log abgelegt
- Inhalt anzeigbar mittels `dmesg` (diagnostic messages)

Gerätedateien

- Der Zugriff auf unser Modul erfolgt in einer Test-Anwendung im User-Mode über eine Gerätedatei.
- Diese befindet sich im Verzeichnis `/dev`
  - Hier `/dev/mm`
- User-Mode: Zugriff auf Geräte/Treiber mit normalen Dateisystemoperationen:
  `int open(const char *pfadName, int openFlag)`
  - Als System Call aus dem User Kontext,
  - Liefert FileDeskriptor oder Fehler (Error < 0),
  - Sowohl für echte Dateien als auch für Geräte.
- Kernel-Mode; Entsprechende Funktion im Treiber:
  `int open(struct inode *i, struct file *fp)`
  - inode: Major, Minor Nummer und Gerätetyp.

Gerätedatei für den Zugriff auf eigenes Modul

- Wir verwenden eine statische Zuordnung mit dem Befehl
  `mknod name type major minor`
  - Hier: `mknod /dev/mm c 241 0`
  - `c` → Character-Device,
    - Neuer Inode im Dateisystem,
  - Inode mit Feld vom Typ dev_t = (241,0),
  - Treibernummer „241“ (Major Number)
  - Gerätenummer „0“ (Minor Number).
- Besser dynamisch (siehe später)



- Das Modul registriert beim Kern ein Char-Device-Objekt auf 241,0

Datentransfer zw. Treiber und Anwendung

- Problem: Treiber darf nicht auf virtuellen User-Mode Speicher zugreifen:
- Getrennte Prozessadressräume → u. U. ist während der Abarbeitung des I/Os ein anderer Prozess aktiv!
- Standardmässig geschieht gepufferte E/A:
  - Treiber muss Daten explizit umkopieren.
  - **`copy_from_user`** `(unsigned long to, unsigned long from, unsigned long len);`
  - **`copy_to_user`** `(unsigned long to, unsigned long from, unsigned long len);`

Hörsaal-Aufgabe 1

- Verwenden Sie als Grundlage das vorgegebene Modul `mm`
- Beim Aufruf von `read` soll das Modul die Segment-Informationen des Aufrufers aus `mm_struct` zurückliefern, siehe Ausgabe unten.
  - Hierzu müssen Sie in `mm_read` Code ergänzen.
- Tipp: Sie können die Daten mit `sprintf` bequem formatieren und zunächst in einem Kernel-Buffer zusammenbauen und am Ende umkopieren `copy_to_user`
- Beispiel-Ausgabe des fertigen Programms:

```
  start_code = 08048000   end_code = 080487F0
  start_data = 08049F08   end_data = 0804A034
   start_brk = 09A5E000    end_brk = 09A7F000
   arg_start = BF96A831    arg_end = BF96A838
   env_start = BF96A838    env_end = BF96AFF5
 start_stack = BF9686F0    down_to = BF9476F0  <--- stack grows downward
```

Dieser Wert steht nicht direkt so in `mm_struct`

Aufgabe 2

- Verwenden Sie als Grundlage das vorgegeben Modul `vma`
- Beim Aufruf von `read` soll das Modul die Liste der VMAs des Aufrufers ausgeben, siehe Ausgabe unten.
  - Einstieg in die Liste ist `mmap` in `mm_struct`.
  - Sie müssen in `vma_read` Code ergänzen.
- Beispiel-Ausgabe des fertigen Programms

```
List of the Virtual Memory Areas for task 'test' (pid=8790)

#VMAs: mm->map_count=17

  1  vm_start=08048000  vm_end=08049000  r-xp
  2  vm_start=08049000  vm_end=0804A000  r--p
  3  vm_start=0804A000  vm_end=0804B000  rw-p
  4  vm_start=0804B000  vm_end=0804C000  rw-p
  5  vm_start=08E79000  vm_end=08E9A000  rw-p
  6  vm_start=B7D81000  vm_end=B7D82000  rw-p
  7  vm_start=B7D82000  vm_end=B7F32000  r-xp
  8  vm_start=B7F32000  vm_end=B7F34000  r--p
```