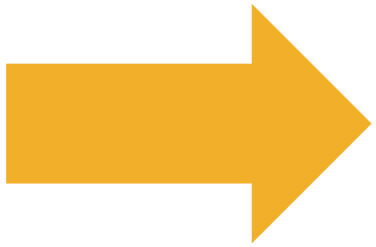
Einführung in die Betriebssysteme

Martin Spörl

Speicherverwaltung

Grundlagen

- zentrales Konzept für parallele Prozessausführung
- speichert alle Informationen eines Programms
  - Code
  - statische Daten
  - dynamische Daten
- besteht aus mehreren Speicherzellen
  - Speicherzellen sind nummeriert => „Adresse“
  - Größe und damit Anzahl der Adressen durch Prozessor festgelegt



Eindeutiges & zufälliges Abfragen mittels Adressen

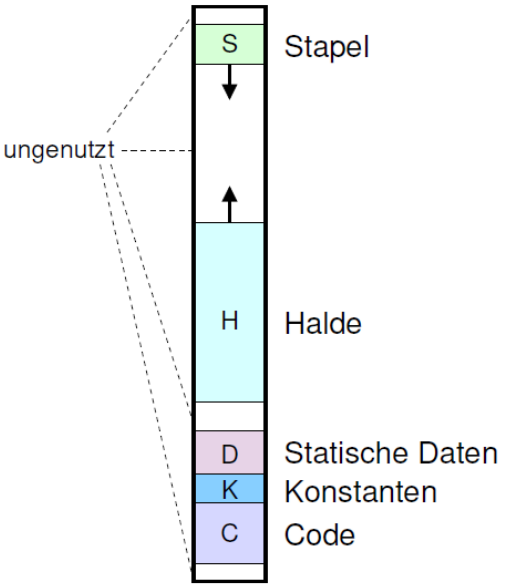
Aufbau des Speichers I

Code

- Programm besteht aus Befehlen
- alle Befehle zusammen bilden den Code
- im Speicher wird der Code in einzelnen Sequenzen abgelegt
- Sequenzen können zufällig im Speicher liegen
  - Aufwand wird aber vereinfacht – man schreibt so nahe wie möglich

Daten

- statische Daten / Konstanten
  - Sind bereits während der Erstellung bekannt
  - Daten die während der gesamten Ausführung vorhanden sind (C: static)
- dynamische Daten (Heap & Stack)
  - werden während der Ausführung definiert & verwaltet



Aufbau des Speichers II

Stack

- „Stapel“
- strukturierter Aufbau & fixe Größe (OS abhängig)
- Daten müssen in umgekehrter Reihenfolge freigegeben werden, wie sie angelegt worden
- wächst „nach unten“

Parameter

Heap

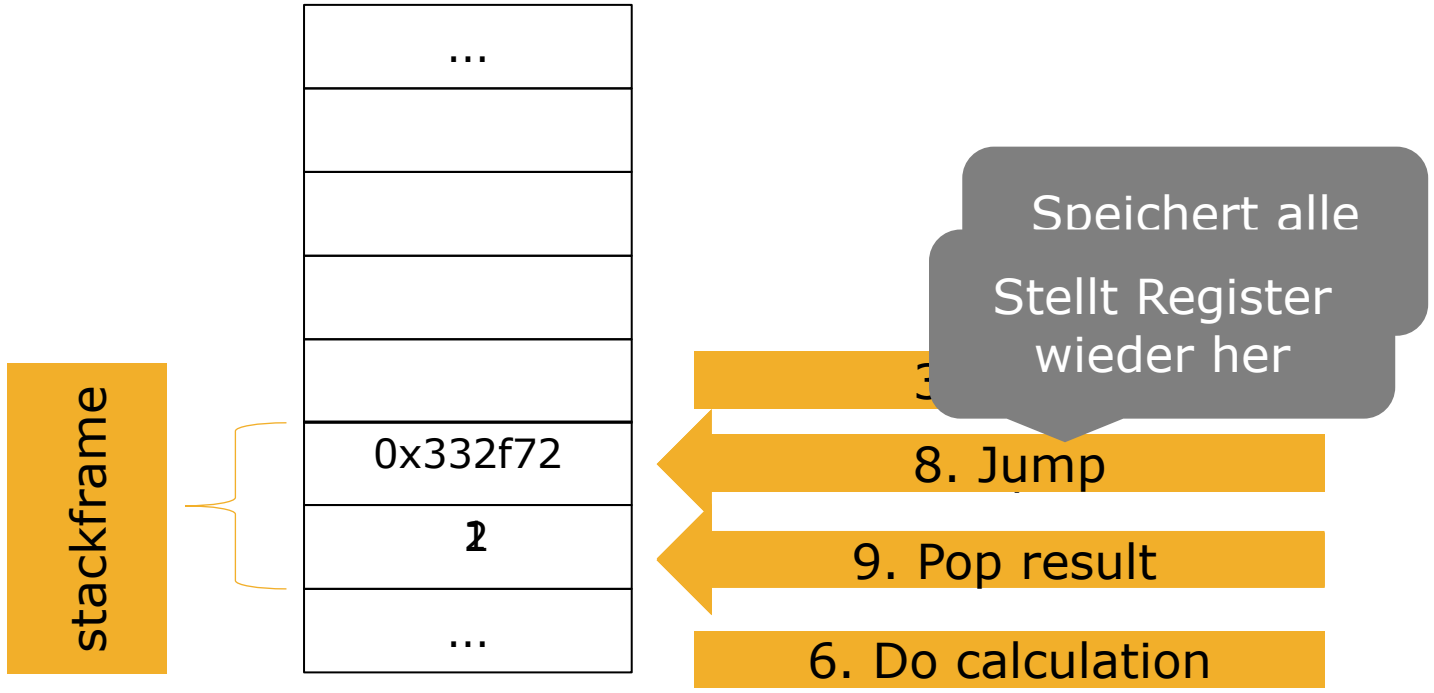
- „Halde“ / „Haufen“
- unstrukturierte Blöcke & variable Größe
- Beliebige Reihenfolge für Freigabe
- wächst „nach oben“

malloc() / calloc() / realloc()

Exkurs: Arbeitsweise von Stack

Stack vs. Heap

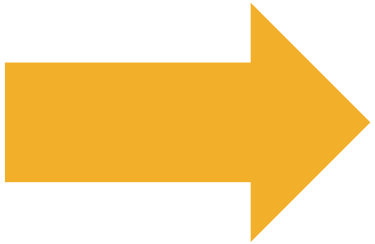
```
int function double(int a){
        return a+a;
}

int a = 1;
int result = double(a);
```



Adressumsetzung I

Wo ist das Problem?

- heutige Rechner führen immer mehrere Programme aus
- jedes Programm braucht Speicher
- Speicher muss früher oder später reserviert werden
- alle Programme teilen sich den gleichen Speicher
- alle Programme können auf **ALLE** Speicherbereiche zugreifen

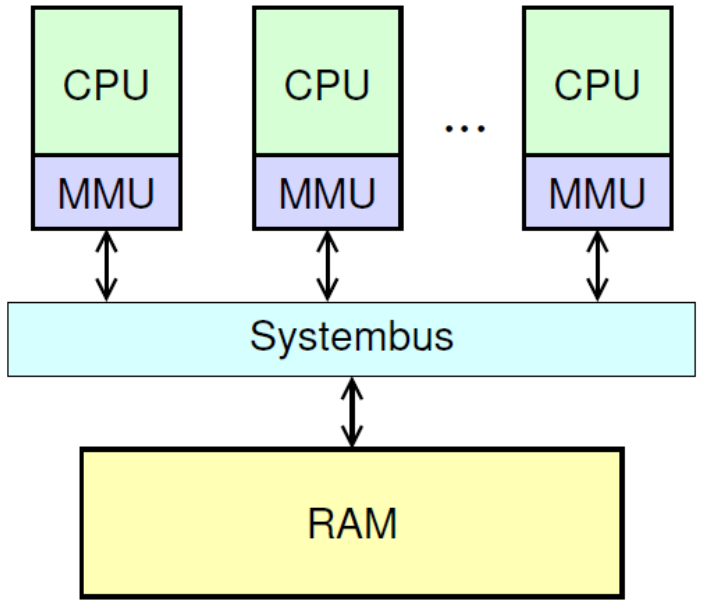


Programmierfehler / böswillige Absichten bringen System in Gefahr!

Adressumsetzung II

Die Rettung – Virtual Memory & Memory Management Unit (MMU)

- verwaltet keinen Speicher!
- übersetzt virtuelle Adressen in physikalische
- kontrolliert Zugriff auf Speicher
- jeder Prozessor hat eigene MMU



MMU - Früher

Umrechnung nach fixer Partitionierung

- Phy. Memory wird in feste Blöcke geteilt (Blöcke unterschiedlich groß)
- pro Prozess wird kleinste ausreichende Partition gesucht
- Hardware definiert einen base register („Startpunkt“)
- Phy. Adresse = virt. Adresse + base register

Nicht genutzter Speicher in einer Partition steht keinem Prozess zur Verfügung
„interne Fragmentierung“

Umrechnung nach variabler Partitionierung

- Memory wird dynamisch in unterschiedlich große Blöcke geteilt
- Hardware definiert einen base register („Startpunkt“) und „limit register“ („Endpunkt“)
- Phy. Adresse = virt. Adresse + base register; if phy. Adresse <= base + limit

Bei mehrfachem Laden und Entladen entstehen Löcher, da nicht jeder Prozess gleich viel Memory braucht
„externe Fragmentierung“

MMU - Heute

Umrechnung nach Seiten (Paging)

- Phy. und virt. Memory wird jeweils in gleich große Blöcke geteilt (Blöcke selbst: unterschiedlich)
  - Größe frei wählbar (typisch: 1KiB, 4KiB, 8KiB)
- Virt. Adresse hat 2 Teile:
  - HSB: virtual page number (VPN)
  - LSB: offset
- Page table ermittelt durch VPN die Page Frame Number (PFN)
- Phy. Adresse = (PFN::offset)

HSB = „High Significant Bits“
LSB = „Least Significant Bits“

Hinweis: Der Arbeitsspeicher wird auch in „Seiten“ eingeteilt – die virtuelle Einteilung ist aber unabhängig davon!
Daher wird der physikalische Speicherabschnitt statt als „Basisadresse“ oft als „Page Frame“ bezeichnet um es abzuheben

MMU Komponenten

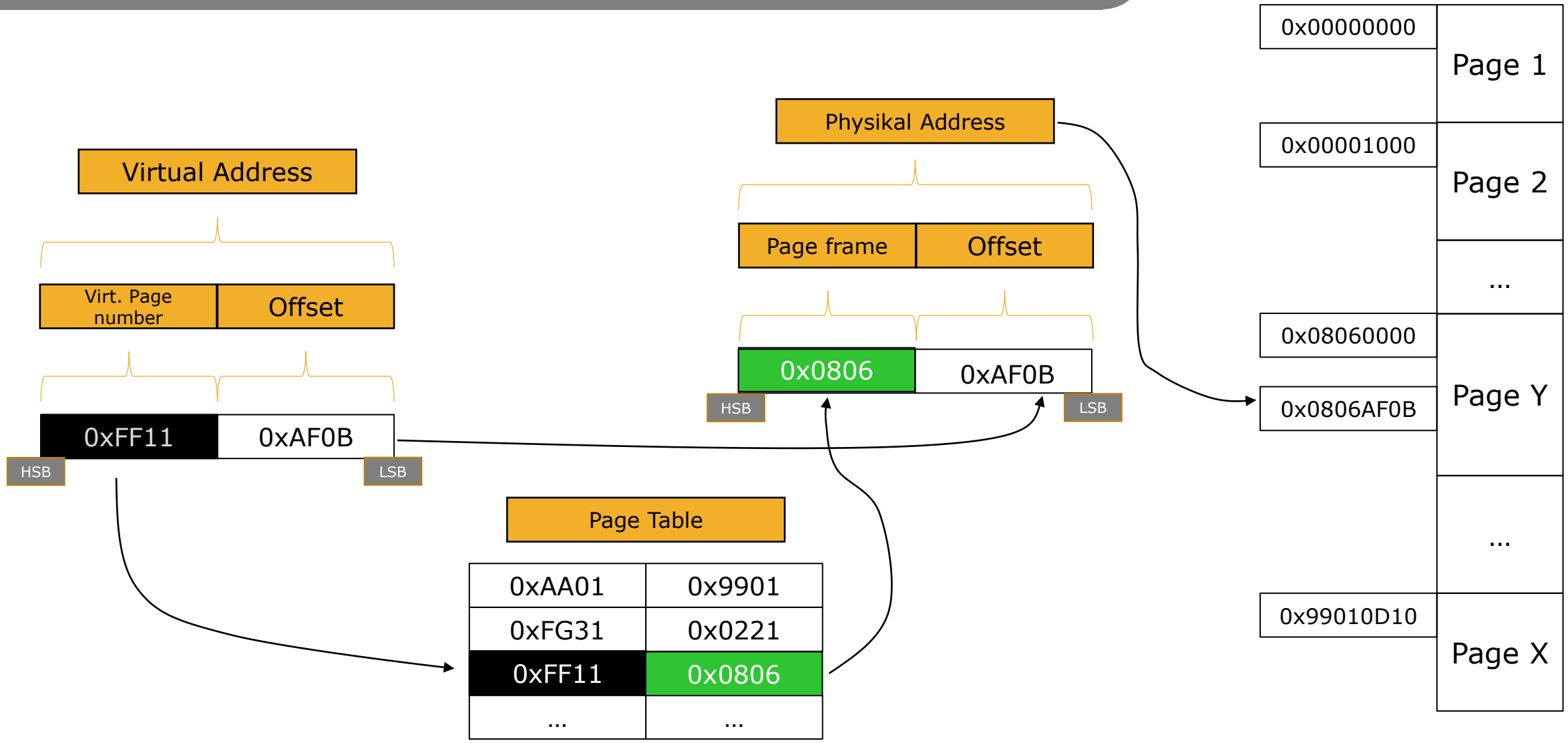
Translation Table (Page Table)

- Im Hauptspeicher
- Speichert diverse Infos
- „Modify Bit“ (Wurde Page verändert?)
- „Reference Bit“ (Wurde Page angefragt?)
- „Valid Bit“ (Kann der Eintrag verwendet werden?)
- „Protection Bits“ (Welche Aktionen – RWE – sind erlaubt?)
- Page Frame Number

Translation Lookaside Buffer (TLB)

- Cache für MMU
- erhebliche Performance Verbesserung!

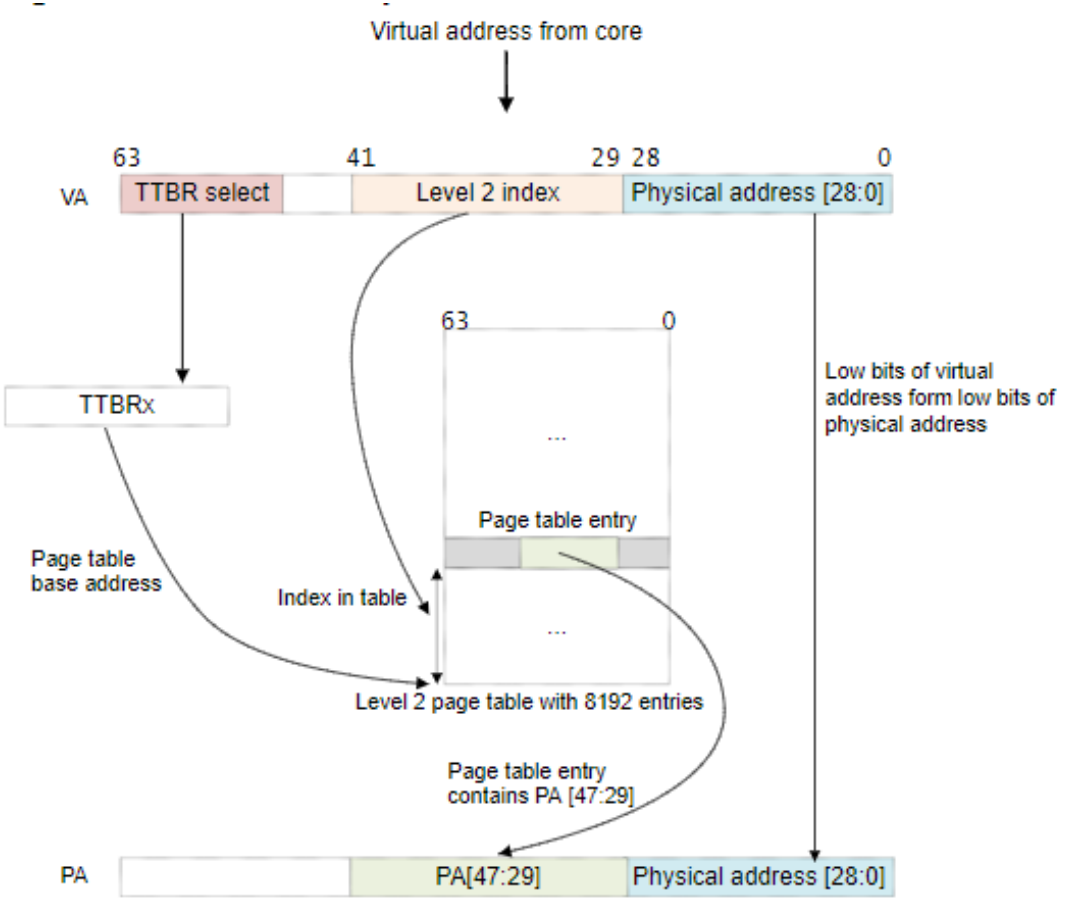
MMU Funktionsweise



Exkurs: MMU in der Praxis

ARMv8 MMU (1 Level; vereinfacht)

- TTBR = Translation Table Base Register
- Virtuelle Adresse besteht aus 3 Teilen
- Mit TTBR wird Basis der Page Table ermittelt
- Mit Level 2 Index wird in der Page Table der Eintrag gefunden
- Page Table Entry + „physical address“ = reale physikalische Adresse



Page Fault

Mögliche Ursachen

- MMU findet keine Übersetzung
- Zieladresse kann gefunden werden, Zugriff ist aber nicht erlaubt
- Schreiben oder Lesen schlug auf Zieladresse fehl

Behandlung

- MMU löst Exception an Prozessor aus
- Prozessor kann weiteres vorgehen entscheiden

Endet fast immer in Exception

Paging vs. Swapping vs. Demand Paging

Paging

- Strategie zur Speichereinteilung
- Speicher wird in Blöcke geteilt
- erlaubt es Memory dynamisch zu reservieren

Swapping

- Kopiert gesamten Prozess Kontext
- Auslagerung auf sekundären Storage (z.b. HDD)

Demand Paging

- „Brücke" zwischen Swapping und Paging
- sorgt dafür dass nicht alle Teile des Prozesses im Memory sein müssen
- Nur selten genutzte Pages werden ausgelagert

Das Gegenteil von Demand Paging ist „Prepaging" – das laden von Speicherblöcken, für inaktive Prozesse, die wahrscheinlich bald wieder aktiv werden

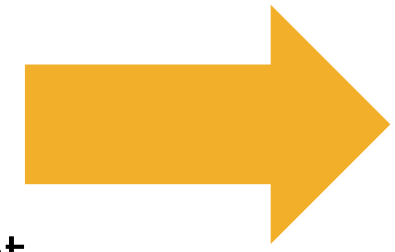
Swapping I

Definition

- Auslagerung von Arbeitsspeicher auf Festplatte
- OS Kern wird nicht ausgelagert!

Vorteil

- schafft mehr Speicher
- mehr Prozesse mit größerem Speicherbedarf können bearbeitet werden

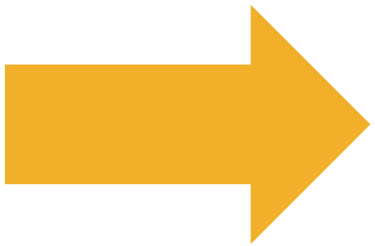


Seitenersetzungstrategien sind nötig!

Swapping II

Gängige Strategien

- Not recently used (NRU)
  - lagert Seiten aus, die lange nicht genutzt wurden und wahrscheinlich auch nicht so bald genutzt werden
- First in, first out (FIFO)
  - die älteste Page wird zuerst ausgelagert
  - oft in Kombination mit Second-Chance-Algorithmus
- Least frequently used (LFU)
  - jede Seite hat Info über Nutzung
  - selten genutzte Pages werden ausgelagert



Kombinationen aus mehreren Strategien in der Praxis genutzt

Demand Paging

Grundlagen

- ermöglicht verschieben von Pages
  - Memory > HDD
  - HDD > Memory
- Ablauf
  - am Anfang sind alle Pages im Arbeitsspeicher
  - Wird Arbeitsspeicher voll, muss Platz gemacht werden
  - einzelne Pages werden ausgelagert
  - Verschiebung durch OS => transparent für Prozess!

Page Fault v2.0

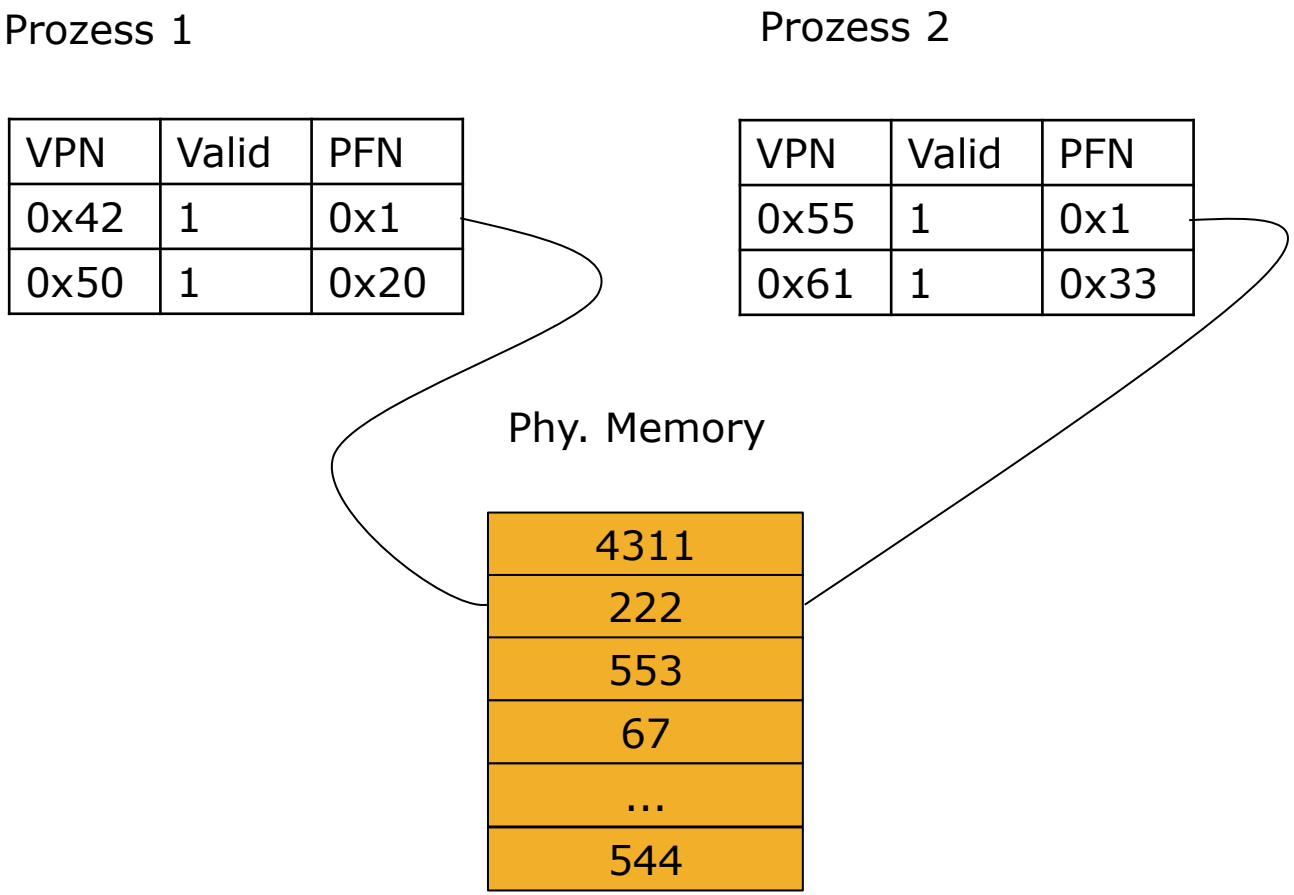
- Problem: Ist Page nicht im Memory => Page Fault wenn Prozess darauf zugreifen will
- Lösung: Page Fault löst „Trap“ aus
  - beim Auslagern wird Eintrag in Page Table ungültig
  - will der Prozess diese Page => Trap wird ausgelöst
  - Trap löst den OS Page Fault Handler aus
  - Routine nutzt den invaliden Eintrag um Daten zu finden
  - Daten werden in Memory geladen
  - Page Table Eintrag wird aktualisiert

Page Sharing

Gemeinsam Speicher sparen...

Grundlagen

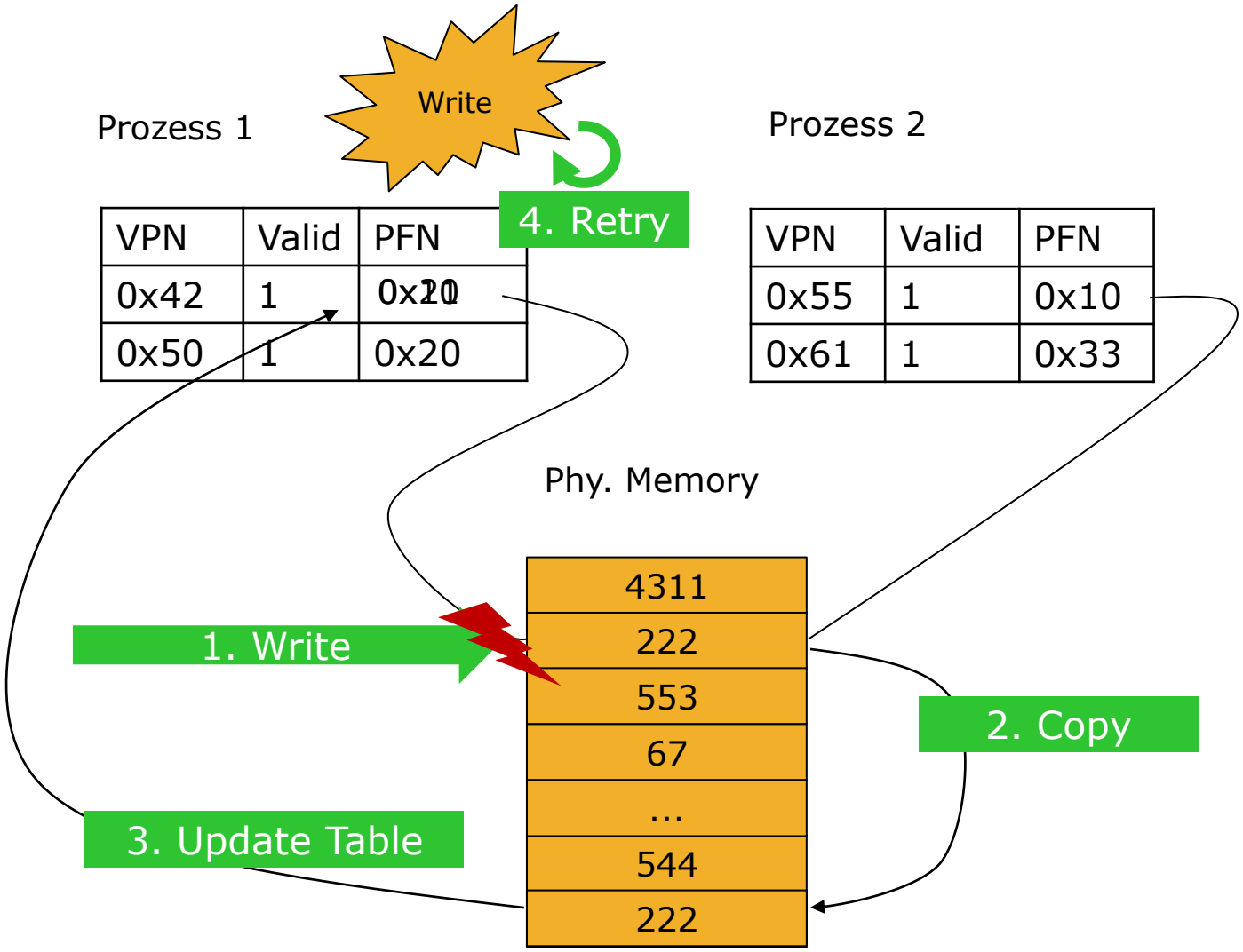
- OS kopiert im Speicher sehr viel
- Programme können gleiche Pages generieren
- gleiche Pages führen zu Redundanzen
- Lösung: Page Sharing
  - MMU verweist auf gleiche Pages, wenn Daten übereinstimmen



Page Sharing – Copy on Write

Und was wenn einer was ändert?

- Idee: Daten nur dann kopieren, wenn sie verändert werden
- Ablauf:
  - Shared Pages werden „read-only“
  - „Write“ löst Trap aus
  - Trap Handler des OS kopiert Page & aktualisiert Page Table Eintrag

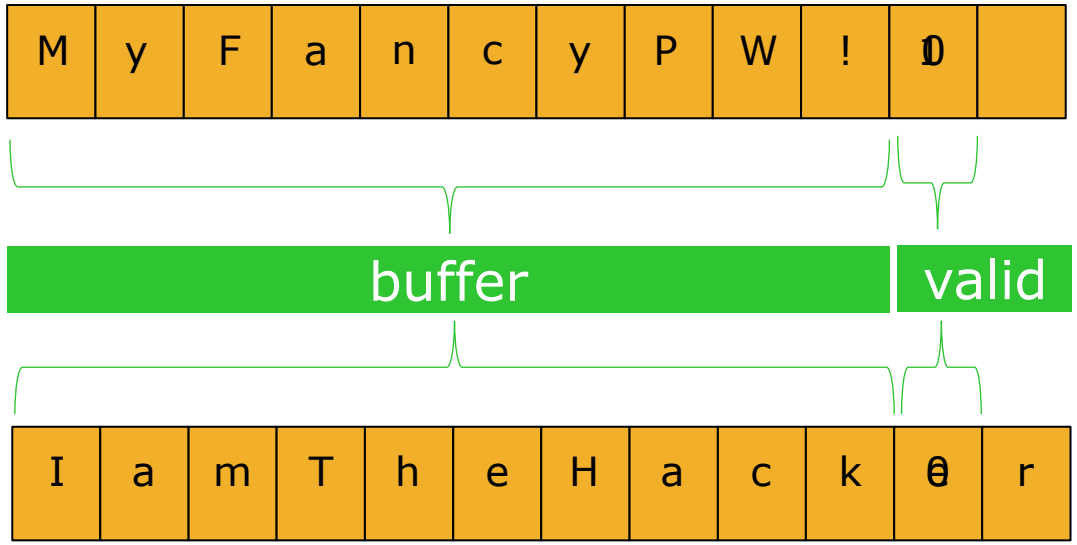


Exkurs: Address Space Layout Randomization

Was ist ein Buffer Overflow?

```
int main(void){
      char buffer[10]; //Annahme 1 Zeichen = 1 byte
       int valid = 0;
       printf(„Please enter your password“);
       gets(buffer);

        if(strcmp(buffer,“MyFancyPW!“) != 0){
                    printf(„this is not the page you are looking for!“);
       }
       else {
                    valid = 1;
      }

      if(valid != 0){
                    printf(„you shall pass!“);
      }
}
```

Input: MyFancyPW!



Input: IamTheHacker

Exkurs: Address Space Layout Randomization

Funktionsprinzip

- Adressbereiche werden zufällig zugewiesen (nicht nebeneinander)
- wird z.b. im Heap angewandt
- sorgt dafür das Variablen nie am selben Platz im Speicher liegen
- erschwert u.a. Buffer-Overflow-Attacken

Bekannte Umgehung: Spraying

- Schadcode wird in großen Mengen dupliziert im Speicher dupliziert
- steigert die Wahrscheinlichkeit doch noch von einer Bibliothek ausgeführt zu werden

Bietet dennoch erhöhten Schutz!