Kapitel 2 - Speicher (CPU):

Virtuelle Speicherverwaltung (VMM): jedem Prozess wird virtuellen Speicher in Form einesvirtuellen Adressraums (Virtual Address Space) zur Verfügung gestellt
Bsp: System ohne Speicherverwaltung; Prozesse A, B, C -> geg: Multiprogramming & Timesharing |32-Bit adressierbarer Speicher mit 1 Byte Kapazität pro Adresse (erlaubt 4GB) | Jedes Programm hat direkten Zugriff auf jede Adresse
->*Szenario1*: OS belegt 1 GB | Prozess A läuft + belegt 2 GB | Prozess B soll gestartet werden + braucht 1,5 GB
Probleme: **1.** Es gibt zu wenig Speicher, um alle Prozesse im RAM zu halten.
2. Prozess B ist abhängig von Prozess A bei der Wahl der Speicheradressen.
Anforderungen: Speicherverwaltung nach Bedarf; kompletter Adressraum für jeden Prozess
->*Szenario2*: OS belegt 1 GB |Prozess A + B belegen jewei 1 GB |1 GB freier Speicher |Prozess C (1 GB) soll started werden
Problem: **3.** Es ist in Summe genug Speicher frei, um Prozess C zu laden, aber nicht an einem Stück (Fragmentierung)
Anforderung: Effizienz bei der Ressourcenverwaltung
->*Szenario3:* Prozess A und B laufen und belegen Speicher | beide Prozesse haben gleiche Speicheranweisungen
Problem: **4**. Die Speicherressourcen der Prozesse sind nicht voneinander abgeschottet.
Anforderung: Speicherbereiche schützen (Protection) --> brauchen virtualisierung von physischen Speicher
Virtueller Speicher (VM): von VMM bereitgestellter Adressraum mit folgenden Eigenschaften:
- Abstrahierung: VM ist in Größe und Auslastung unabhängig vom physischen Speicher.
- Schutz: Jeder Prozess bekommt seinen eigenen privaten und isolierten Adressraum.

Realisiert wird VM unter folgenden Anforderungen:
- Transparenz: VMM ist aus Sicht des Prozesses unsichtbar. Jeder Prozess hat Zugriff auf gesamten Adressraum.
- Effizienz: Fragmentierung (räumlich) und Overhead (zeitlich) werden minimiert

erster Versuch Virtualisierung: Suchen Mechanismus mit den Eigenschaften Effizienz, Schutz und Transparenz.
->Betriebssystem stellt sicher, dass kein Prozess zugewiesene Speicherbereiche überschreitet
Annahmen: **1**. Die Daten eines Prozesses liegen fortlaufend im Speicher | 2. Der virtuelle Adressraum (Virtual Address Space, VAS) ist kleiner als der physische Adressraum (Physical Address Space, PAS). | 3. Jeder VAS hat die gleiche Größe. | 4. Speichergrößen: 16 KB (VAS) und 64 KB (PAS)
Entkopplung von VAS und PAS: Über MAP von VAS mit Offset in PAS.
-> Von PAS zu VAS: benötigen Möglichkeit von physischen Adrs in virtuelle Adrs umzurechnen -> Ansatz für Adrsübersetz
Base& Bounds: - stehen für Versatz (Base) + Grenzen (Bound/Limit) bei Umrechnung: *Physical Adrs = Virtual Adrs + Base*
- erlauben es, den Adressraum eines Prozesses auf beliebige Stellen im PAS abzubilden
- Register, die auf Software- oder Hardware-Ebene bereitgestellt werden (Ansatz: Dynamic Relocation)

Bemerk: Bounds-Register dient als Schutzmechanismus für Bereichsüberschreitungen | Zugriffe außerhalb des zugewiesenen Bereichs führen in Regel zu Ausnahmen und entsprechende Prozesse werden vom OS beendet
Anforderungen von Adressübersetzung mittels Base & Bounds:
an Hardware: Priviligierter Modus |Base- und Bounds-Register je Prozessor |Mechanismen für Übersetzung und Einhaltung der Bereichsgrenzen |Instruktionen zur Register-Aktualisierung |Möglkeiten zur Ausnahmebehandl
an OS: -Speicherverwaltung: weist Speicher für neue Prozesse zu und gib Speicher von beendeten Prozessen frei
Freier Speicher wird in einer Datenstrukturverwaltet
-Base & Bounds Verwaltung: Aktualisiere Register bei einem Context Switch
-Ausnahmebehandlung: Spezifiziere was passiert, wenn eine Exception geworfen wird
Zwischenstand von aktuellen System zu Anforderungen:
-*Abstrahierung*: Virtualisierung des Speichers kann mittels Base & Bounds implementiert werden.
-*Schutz*: Bereichsüberschreitungen von Bounds erkannt und abgefangen -> Ausführung erfolgt im Kernel Mode
-*Effizienz (Laufzeit)*: halten den Mehraufwand durch Hardwareunterstützung gering
-Transparenz: von Prozess her ist VMM nicht vorhanden -> er kennt nur eigenen VAS (beginnt bei 0 KB bis .....)
Probleme: **1**-Wir modellieren VAS < PAS -> in Realität wollen wir aber VAS > PAS |**2**-Daten eines Prozesses liegen fortlaufend im Arbeitsspeicher. zsm mit der ersten Vereinfachung schwer zu bewerkstelligen -> wenn: VAS > PAS gilt, dann passen Prozesse nicht mehr in Speicher | **3**-Jeder VAS hat gleiche Größe ->wollen eine flexible Lsg
4-Interne Fragmentierung wird nicht beachtet
Interne Fragmentierung: ist Effekt, bei dem Speicherbereiche innerhalb logischer Speichereinheiten nicht genutzt werden
->Eine effiziente Speicherverwaltung versucht dies zu vermeiden oder zumindest zu minimieren
Bsp: den Prozessen wird zu viel Speicher zugeordnet, der dann nicht von anderen Prozessen verwendet werden kann
Bemerk: Hardware, die zusammen mit Prozessor auf Chip sitzt und für Speicherverwaltung zuständig ist, heißt Memory Management Unit (MMU) | Base&Bounds, auch dynamische Speicherverwaltung genannt, da virtuelle Aderssen zur Laufzeit übersetzt werden -> Statische Speicherverwaltung ist eine Technik, mit der virtuelle Adressen des Prozesses vor Ausführung überschrieben werden.
Segmentierung: Annahme 2+3 wird verworfen; VAS jetzt beliebige Größe. Ziel: Fragmentierungsproblem lösen.
Generalisierung des Base&Bounds Systems: - einfache Idee um das Fragmentierungsproblem zu lösen
- für jedes logische Segment (Code, Heap, Stack) des Adressraums legt man Base & Bounds Register an

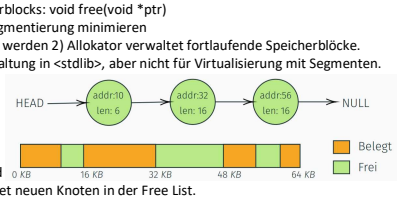
->Um Segmente im System identifizieren zu können, benutzen man paar Bits des Adressraums -> Für 3 Segmente benötigt man 2 Bits, die dann als ID fungieren: geg. 14 Bite Array: 13+12 -> Segment ID: 11 bis 0 -> Offset
->Offset zeigt, ob die Adresse innerhalb der Bounds liegt (Wert kleiner als 2048, dann Adresse erlaubt)

```
# SEG_MASK = 0x3000; SEG_SHIFT = 12; OFFSET_MASK = 0xFFF
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT # (Oberste zwei Bits ermitteln)
Offset = VirtualAddress & OFFSET_MASK # Hintere 12 Bits ermitteln
if (Offset >= Bounds[Segment]): # Offset größer als erlaubte Bounds
  RaiseException(PROTECTION_FAULT) # Exception werfen
else: # Offset ist erlaubt
  PhysAddr = Base[Segment] + Offset # Physische Adresse ist Base + Offset
  Register = AccessMemory(PhysAddr) # Zugriff auf Physische Adresse
```

- In unserem Beispiel geht ein Segment des Adressraums verloren.
- Der virtuelle Adressraum wird durch den Verlust der zwei Bits stark reduzier

Funktionserweiterungen für Segmentierung: Heap + Stack wachsen dynamisch in unters Richtungen. Code ist statisch
Wachstumsrichtung (*Growth Direction*): einzelnes Bit (Flag) gibt an, ob die Richtung positiv ist oder nicht.
Geteiltes Code-Segment (Code Sharing): Manche Segmente des Speichers sind einheitlich für verschiedene Prozesse und teilbar, dann benötigt man Schutzmechanismen, die über Schreib-, Lese- und Ausführungsrechte entscheiden
Externe Fragmentierung (*External Fragmentation*): in Summe ist genug freier Speicherplatz vorhanden, um Anfragen bearbeiten zu können -> Speicherplatz ist aber nicht am Stück frei
->wenn Fragmentiert, dann abwechseln Frei und Belegt /-> wenn defragmentiert, dann erst Belegt und danach Frei

Dynamische Speicherverwaltung: Speicherverwaltungssysteme (Ohne Virtualisierung):
Free Space Management (FSM): Strategien zum Allozieren und Freigeben von Speicher -> Ziel: externe Fragmentierung zu minimieren ->Strategien relevant, wenn der allozierte Speicherbereich variabler Größe sein kann
Fallbeispiel Heap: Schnittstelle konzepieren, die für Verwaltung des Heaps fungiert
->Geforderte Funktionen: Allozieren eines Speicherblocks für Objekt der Größe size: void *malloc(size_t size)
->Freigeben des von ptr referenzierten Speicherblocks: void free(void *ptr)
-> gefordert: soll schnell sein und (externe) Fragmentierung minimieren
Annahmen: 1) Allozierter Speicher kann nicht verlagert werden 2) Allokator verwaltet fortlaufende Speicherblöcke.
-> Annahme 1 realistisch für dynamische Speicherverwaltung in <stdlib>, aber nicht für Virtualisierung mit Segmenten.
Free List (Verkette Liste): Einträge (address, length) beschreiben freie Speicherbereiche im Heap
Operationen um mit Free List effizent zu Arbeiten:
-***Speicherbereiche aufteilen (Split)***: bei Aufruf von **malloc()** wird passender Knoten ausgewählt und Speicherbereich in zwei Teile geteilt -> 1Teil wird an die rufende Instanz zurückgegeben, 2Teil bildet neuen Knoten in der Free List.



-***Speicherbereiche verbinden (Coalesce)***: bei Aufruf von ***free()*** muss entsprechende Speicherbereich wieder in Free List eingegliedert werden /wenn 2 benachbarte Knoten im Speicher angrenzend, so werden sie verschmolzen (coalesce)
Fragen: - Wo speichern wir die Free List?
- Wo speichern wir Infs über Größe der allozierten Blöcke? z.B. Für int *ptr = malloc(sizeof(int) * 1000);
- Wie können wir bei Schnittstelle free(void *ptr) sicherstellen, dass nur der richtige Speicher freigegeben?

=> Man braucht Header der Free List (typedef struct { int size; int magic; } header_t;)
Header: viele Allokatoren reservieren zusätzlich zum geforderten Speicher extra Informationen in Header-Block, der in der Regel direkt vor dem ausgegebenen Speicher liegt
Bemerk: magic (Magic Number) hilft bei Integritätstests |bei Aufruf von malloc() von N Bytes werden N + sizeof(header) Bytes alloziert | bei Aufruf von free(void *ptr) kann mit einf. Zeigerarithmetik berechnet werden, wo Header beginnt: header_t *hptr = (header_t *) ptr - 1;
Speichern der Free List: FreeList wird in freien Speicherblöcke im Heap eingebettet ->Bsp: wollen 4 KB Speicher verwalten
->Dafür muss die Liste initialisiert werden, die zunächst einen Eintrag folgender Art haben soll:

Free List für Verwaltung von 4KB Speicher:

```
typedef struct __node_t { int size; struct __node_t *next; } node_t; // Listeneintrag
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,MAP_ANON|MAP_PRIVATE, -1, 0); // Initialisiert Liste
head->size = 4096 - sizeof(node_t); // Größe: 4KB
head->next = NULL;
```

Strategien für Knotenauswahl für Free List:
Best-Fit: Durchsuche Free List nach allen Blöcken, die groß genug, um N Bytes zu halten. Gib kleinsten der Blöcke zurück.
-> Intuition: Minimiere Verschnitt (Waste) -> Problem: vollständige Suche nötig
Worst-Fit:Durchsuche Free List nach allen Blöcken, die groß genug, um N Bytes zu halten. Gib größten der Blöcke zurück.
->Intuition: mögl viele große Blöcke intakt lassen (anstatt kleiner) ->Problem: vollständige Suche + big Verschnitt
First-Fit: Durchlaufe Free List bis zum ersten geeigneten Knoten und gib diesen zurück.
->Intuition: sehr schnell ->Problem: Anfang der Free List wird überproportional mit kleineren Objekten belegt
Next-Fit: Erweit First-Fit um Zeiger auf den Eintrag, der als letztes verwendet. Nimm ihn als Startknoten beim nächsten Aufruf von malloc() | ->Intuition: bessere Lastverteilung und schnell ->Problem: abhängig vom Szenario
In Praxis: -**Slab Allocator für Kernel Heap in Linux/Solaris**: eigener Cache-Speicher für Anfragen oft vorkommender Größen (z.B. Locks, Inodes). Bei anderen Anfragen wird Standard-Allokator vom System genutzt. Dadurch Fragmentierung fast ausgeschlossenund sehr schnell. Allerdings zusätzliche Ressourcen nötig.
-**Binary Buddy Allocator**: unterteilt Speicher in Blöcke (Größe 2^N). Freier Speicher wird rekursiv durch zwei geteilt, bis Block gerade groß genug ist. Einfaches Coalescing (Buddies unterscheiden sich nur in einem Bit und Wertigkeit durch Hierarchie gegeben), aber gibt interne Fragmentierung. Mittlerweile meist balancierte Binärbäume/Bäume mit Halbordnungen (Liste durchgehen dauert sehr lang, daher schlechte Skalierbarkeit)
Bemerk: Komplexe Systeme zur Verwaltung freien Speichers immer nötig, wenn Speicherblöcke variabler Größe alloziert werden müssen. Wahl des richtigen Allokators abhängig vom Anwendungsfall und Arbeitslast.

Zwischenstand Segmentierung: Speicherverwaltung für beliebige Größen mittels Segmentierung lösbar. Aufteilung in logische Segmente mit variabler Größe führt aber schnell zu Fragmentierung (Schlimmer, je länger das System läuft).
=> Aufteilung des Speichers in Stücke gleicher Größe, die fest vorgegeben ist. (Paging)
Paging: Methode zur Speicherverwaltung und ermöglicht effiziente Virtualisierung; Ansatz unterteilt virtuellen Adrsraum in gleich große Blöcke, die man Seiten nennt (Page) -> das Design verhindert externe Fragmentierung
-> Interne Fragmentierung wird durch richtige Wahl der Page Size reguliert.
-> physische Adressraum wird auch in Blöcke unterteilt (Kacheln, Page Frames).
-Von Pages zu Frames: eine Menge Pages in Prozessen + eine Menge von Page Frames im physischen Adressraum
-> irgendwo müssen Informationen vorgehalten werden, was wo liegt -> PageTables
Page Tables: Betriebssystem benötigt zum Übersetzen der VirtuellenAdresse (VA) in physikalischeAdresse (PA) eine Datenstruktur, welche für Pages eines Prozesses die zugehörigen Page Frames hält. Diese Datenstruktur heißt Page Table -> für jeden Prozess wird eine solche Page Table angelegt
Ablauf Adressübersetzung: **1**-VA in Abhängigkeit von Page Size in Virtuelle Page Number (VPN) und Offset unterteilt | **2**-Für VPN wird in Page Table physische Page Frame Number (PFN) ermittelt |3-PA ergibt sich auf PFN und Offset von VA
Page Table Size: abhängig von Größe des VAS und Page Size. Beispiel: 32bit VAS mit 4KB Pages: 12 Bit Offset, 20 Bit VPN -> 2^20 mögliche| Übersetzungen => 4Byte pro Eintrag, dann macht das Größe von 4MB pro Page Table (bei 3600 Prozessen: 14,4GB!). Das ist schlecht.
Page Table Entry (PTE): PFN, Present Bit (Page im RAM?), Dirty Bit (Page modifiziert?), Reference/Access Bit (Page wurde verwendet?). Ist aber je nach Architektur unterschiedlich, x86 hat z.B. noch Superviser/Write-Bits etc.
Lineare Page Table: Wird durch VPN indiziert (D.h. pro VPN hat man in Page Table eine PFN).
Adressübersetzung (2x Mem-Access):

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT # VPN auslesen
PTEAddr = PTBR + (VPN * sizeof(PTE)) # PTE-Adresse ermitteln
PTE = AccessMemory(PTEAddr) # PTE auslesen (1. Speicherzugriff!)
if (PTE.Valid == False): # Kein gültiger Zugriff
  raise Exception(SEGMENTATION_FAULT)
elif (CanAccess(PTE.ProtectBits) == False): # Zugriff nicht erlaubt
  raise Exception(PROTECTION_FAULT)
else: # Gültiger Zugriff
  offset = VirtualAddress & OFFSET_MASK # Offset ermitteln
  PhysAddr = (PTE.PFN << PFN_SHIFT) | offset # Physische Adresse für Daten
  Register = AccessMemory(PhysAddr) # Daten auslesen (2. Speicherzugriff!)
```

Bemerk: -Adressübersetzung via Paging benötigt weiteren Speicherzugriff, muss nämlich zusätzlich auch PTE laden
-da Speicherzugriffe auf Speicher im Allgemeinen sehr teuer sind, ist Paging damit deutlich langsamer. (Beschleunigen durch Hardware-Caching) | - gibt noch 2 Probleme: Speicherbedarf und Laufzeitverhalten
Zwischenstand: -***Paging*** -> verhindert externe Fragmentierung |unterstützt wie Segmentierung einen beliebig großen VAS |ist flexibel in der Wahl der Page Size | ist prinzipiell einfach.
-***Probleme*** -> Effizienz (räumlich): Page Tables sind in bisherigen Form (linear, zufälliger Zugriff) sehr groß!
-> Effizienz (zeitlich): Paging teurer, da für jeden Speicherzugriff ein Zusatzzugriff auf Page Table entste
Zeitl. Effizienz: Häufig Zugriffe auf gleiche Speicherbereiche -> häufig genutzt Übersetzungen cachen (in Hardware-Cache)
Hardware-Caching: oft genutzte Adressübersetzungen werden zwischengespeichert (um Speicherzugriffe zu verringern)
->daraus folgt: **Translation Lookaside Buffer (TLB)**: ist Cache und sitzt als Teil der MMU mit der CPU auf einem Chip.
->Aufgabe: vorhalten häufig genutzte VPNs, um teure Zugriffe auf die Page Table deutlich zu reduzieren
Funktionsweise des TLB:

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT # VPN ermitteln
(Success, TlbEntry) = TLB_Lookup(VPN) # TLB Eintrag auslesen
if (Success == True): # Eintrag vorhanden (TLB Hit)
  if (CanAccess(TlbEntry.ProtectBits) == True): # Zugriff erlaubt
    Offset = VirtualAddress & OFFSET_MASK # Offset ermitteln
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset # PFN ermitteln
    Register = AccessMemory(PhysAddr) # Zugriff auf PAS
  else: # Zugriff nicht erlaubt
    raise Exception(PROTECTION_FAULT)
else: # Eintrag nicht vorhanden (TLB Miss)
  PTEAddr = PTBR + (VPN * sizeof(PTE)) # PTE-Adresse ermitteln (Aus Page Table Register)
  PTE = AccessMemory(PTEAddr) # PTE aus PAS auslesen
  if (PTE.Valid == False): # PTE nicht gültig
    raise Exception(SEGMENTATION_FAULT) # Segmentation Fault
  elif (CanAccess(PTE.ProtectBits) == False): # Zugriff nicht erlaubt
    raise Exception(PROTECTION_FAULT)
  else:
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits) # Eintrag > TLB
    RetryInstruction() # Nochmal von vorne (jetzt ist Entry in TLB)
```

->Durch **Spatial Locality** wird die Performance weiter erhöht. (Ein Page hält mehrere Elemente)
TLB als Cache: - ist auf Geschwindigkeit getrimmt -> deshalb werden kleine TLBs als **Content Addressable Memory (CAM)** implementiert, bei dem alle Speicherzellen parallel geprüft werden können -> unter Annahme, dass die meisten Speicherzugriffe in **TLB Hit** resultieren, können so die Kosten von Paging gering gehalten werden.
Ablauf bei TLB Miss: -***in Hardware*** (z.B. x86): Prozessor sucht automatisch PTE und bringt ihn in TLB -> falls PTE nich gefunden wird, wirft HW PageFaultException, für welche dann das Betriebssystem zuständig ist
-***TLB in Software*** (z.B. MIPS R10k/SPARC v9 von Sun): **1**-Wenn TLB Miss, dann wirft Hardware eine Exception
2-aktuelle Kontrollfluss wird unterbrochen und Prozessor geht in Kernel Mode und dann zu Trap Handler
3-entsprechende Eintrag in Page Table wird gefunden und der TLB wird aktualisiert
4-HW versucht dann Befehl auszuführen, die den TLB Miss ursprünglich verursacht hat.
-> Vorgehen vergleichbar, nur dass man sich in Software selbst um die Suche und Aktualisierung kümmern muss
Einträge eines TLB: Typischer hat 32, 64 oder 128 Einträge und ist komplett assoziativ, dh. es können Übersetzungen an beliebigen Stellen im TLB vorliegen können -> durch Parallelisierung in Hardware werden Einträge schnell gefunden
Table Entry: VPN, PFN |Valid Bit (Valide Übersetzung) |Protection Bit(s) (Wie darf zugegriffen werden) |Dirty Bit (Page modifiziert) |ASID (Address Space Identifier) ->für Context Switch muss Adressraum identifiziert werden
TLB im Kontext von Prozessen: TLB enthält Übersetzungen, die nur für bestimmten Prozess gültig sind
->bei Context Switch muss HW oder OS sicherstellen, dass nächste Prozess nicht falsche Übersetzungen nutzt
Ansätze: -**Flush**: Leere den TLB bei einem Context Switch, bevor der neue Prozess startet. (einfach, aber teuer!)
Address Space Identifier (ASID): separater Eintrag im TLB, der den VAS eines Prozesses identifiziert
-> ASID ist ähnlich zu Process Identifier (PID), nutzt aber weniger Bits
Cache Replacement Policy: regelt welche Einträge ersetzt werden sollen, wenn der TLB voll ist, sobald ein neuer Eintrag
2 Strategien: LRU und RR
LeastRecentlyUsed(LRU): -wähle d am längsten nicht verwend Eintrag im TLB + ersetze ihn durch new benötigten Eintrag
- Annahme zeitli Lokalität: Wahrskeit hoch, dass Daten die gerade in Benutzung, auch in Zukunft benutzt werden
- Annahme ist sehr schlüssigt und führt zu Problemen, wenn nicht zutrifft -> Worst Case: Thrashing

Random Replacement (RR): -wähle zufälligen Eintrag im TLB und ersetze ihn durch den neuen benötigten Eintrag
- Trifft praktisch keine Annahme über das Zugriffsverhalten eines Prozesses.
- Ansatz ist stochastisch motiviert -> er umgeht das Szenario Thrashing des LRU-Ansatzes

TLB Thrashing: beschreibt Phänomen, das auftreten kann, wenn sogenannte Working Set eines Prozesses die Größe des TLB übersteigt -> **Working Set** beschreibt die Menge der von einem Prozess aktiv verwendeten Pages.
-> Menge der aktiv verwendeten Pages übersteigt Cache Size. Dann gibt es nur Cache-Misses
Erste Idee um Speicherhunger von Paging zu lösen: Größe der Page Table hängt unmittelbar von Anzahl der Pages ab
-> man kann also die Größe der Page Table durch die Verwendung von größeren Pages reduzieren