Betriebssystemtechnik
Zusammenfassung des Wichtigsten

Sommersemester 2016

Prof. Dr. Schröder-Preikschat

Autor:

- Christian Strate

Inhaltsverzeichnis

0 Organisatorisches

- mikrokern-ähnliches OS in den Übungen

Note:

Bei dieser Zusammenfassung handelt es sich um eine inoffizielle Mitschrift von Studenten. Entsprechend sind weder Garantie auf Vollständigkeit noch auf Korrektheit gewährleistet.

1 Einleitung

Schutz vor Prozessen in räumlicher Hinsichtlich:

- Angriffssicherheit (security) (Immunität, Einbruch in Adressraum, Schutz einer Entität vor seiner Umgebung)
- Betriebssicherheit (safety) (Isolation, Ausbruch aus Adressraum, Schutz einer Umgebung vor einer Entität)

Unterscheidung realer, logischer, virtueller Adressraum

Segmentierung von Programmen:

- Maschinenprogrammebene - Immunität/Isolation durch HW (MMU $\rightsquigarrow$ logisch, MPU $\rightsquigarrow$ physikalisch)
- Programmiersprachenebene - Immunität/Isolation durch SW

Überschreitung der Adressraumgrenzen:

- durch Wechsel der Schutzdomäne
  - prozedurbasiert: Syscall, Kontrollflussfortsetzung im anderen Adressraum
  - koroutinenbasiert: Nachrichten, Kontrollflussfortsetzung hin zum anderen Adressraum
- durch Mitbenutzung von Adressraumbereichen
  - Datenverbund
  - Gemeinschaftsbibliothek

2 Systemaufruf

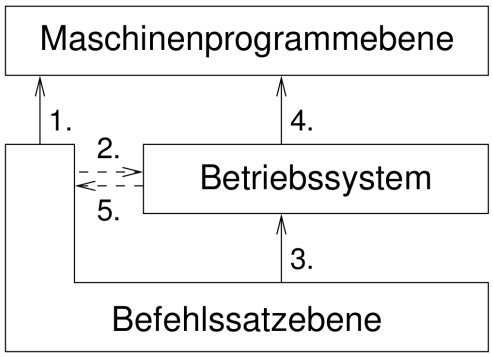


Abbildung 1: Teilinterpretation

1. Interpretation des Maschinenprogramms durch Befehlssatzebene

2. Aussetzen der Ausführung (Ausnahmesituation, Programmunterbrechung) $\mapsto$ Trap um bspsw einen neuen Kontrollfaden zu erzeugen, starten des Betriebssystems. Der HW-Befehlssatz kennt den Befehl nicht, prepending 0b1010 signalisiert "unbekannter Befehl", Befehlssatz wird durch Umschaltung erweitert. OS löst den Befehl auf.

3. Interpretation der Betriebsystem-Programme

4. Interpretation des zuvor unterbrochenen Maschinenprogramms durch das Betriebssystem.

5. OS instruiert die Befehlssatzebene die Ausführung des Programms von dem 4. Punkt wieder aufzunehmen.

- CD80 - syscall
- CF - return

Abstraktion von Betriebssystemabschottung - fig. 2:

- Ortstransparenz: Sowohl für den Aufrufer durch den Aufgerufenen, als auch für die Systemfunktion durch den Aufrufzuteiler.
- Entkopplung des Maschinenprogramms von Programmen des Betriebssystems
- Zugriffstransparenz durch Zugiffsmethoden, die die "Pointer-Auflösung" in den anderen Adressräumen vornimmt - erreicht durch Erweiterung (Zugriffsmethoden) - fig. 3

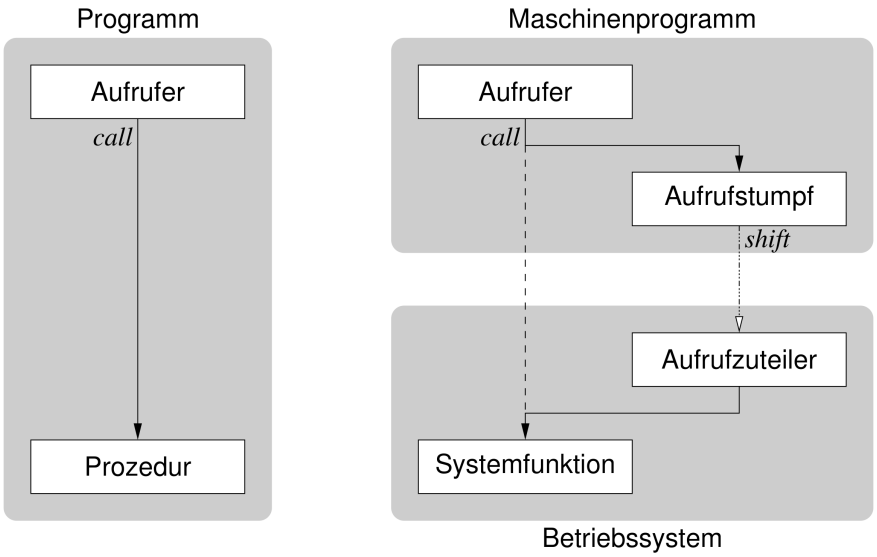


Abbildung 2: Prozedur- vs. Systemaufruf
Der Systemcall ist nicht direkt möglich, sondern bedingt durch den Adressraumschutz eine Indirektion (call + shift, der die Grenze überwinden kann mittels Trap+Traphandler), linkes Bild eher lib-OS (selber Adressraum ⇒ inlining sinnvoll)

Ausnahme erkennen:
Unterscheidung normaler Funktionsergebnisse von Ausnahmefällen:

- verschiedene Wertebereiche im Rückgaberegister (%eax), betriebssystemseitig einfach.
- Übertragsmerker (Flag) im Statusregister, Manipulation des Stack-Frames des Systemaufrufs so, dass der Merker bei Rückkehr die Ausnahme(1) oder Normalfall (0) signalisiert, größerer Overhead, aber konsequent hinsichtlich des Befehlssatzes als Befehlssatzebenenerweiterung

Zustandssicherung - Systemaufrufzuteiler im OS Xunil: Auf dem Stack:

1. Statussicherung der Befehlssatzebene (INT)
2. Statussicherung Betriebssystem (Xunil)
3. Aufruf Systemfunktion des Kerns

Der Aufruf der Systemfunktion erfolgt über die Sprungtabelle, bei der für jeden opcode ein eigener Funktionsname steht.

- keyword asmlinkage im C-Code (vor der Funktionsdeklaration) bittet den gcc, die Funktionsparameter auf dem Stack zu erwarten und nicht über Register.

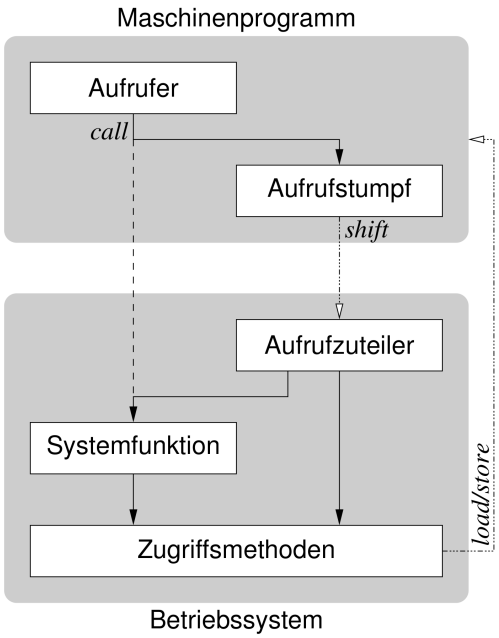


Abbildung 3: Abstraktion von Maschinenprogrammabschottung

```
1 .macro sc scn
2    movl \scn, %eax     ;pass system call number
3    int $128            ;cause software interrupt
4 .endm
```

Listing 1: Assembler-Macro-Beispiel

Primitiv- vs Komplexbefehl:

- Primitivbefehl (RISC-artig) Zeug vor int
  - Prozessstumpf referenziert
  - (+/-)Werteübergabe von Operanden im Maschinenprogramm
  - (+/-)dynamische Operandenauswertung (Laufzeit)
  - (-)begrenzte Operandenanzahl durch Prozessorregistersatz ⇒ effiziente Operanden sind von der Registeranzahl abhängig
  - (-)betriebssystemseitig bestenfalls teilweise Zustandssicherung
  - (-)maschinenprogrammseitiger Mehraufwand zum Operandenabruf
- Komplexbefehl (CISC-artig) Zeug nach int
  - keine Register werden angefasst

  - beliebig viele Operanden sind effizient verwendbar
  - Compiler legt Referenz ab und Kernel dereferenziert (registerlos)
  - entspricht (statischem) Befehlsformat der Befehlssatzebene
  - (+)Auswertung der Operanden erfolgt entsprechend statisch und registerlos zur Assemblier-/Bindezeit
  - (+)kompakte Kodierung der Systemaufrufe
  - (+)betriebssystemseitig vollständige Zustandssicherung
  - (+/-)statische Operandenauswertung (Assemblier- oder Linkzeit)
  - (-)Referenzübergabe von Operanden im Maschinenprogramm
  - (-)betriebssystemseitiger Mehraufwand zum Operandernabruf
  - (-)Die Daten hinter der Referenz in den Userspace, die vom Kernel aufgelöst werden, können von anderen Threads bereits manipuliert worden sein, wodurch verschiedene Lesevorgänge unterschiedliche Ergebnisse liefern können.

- Inlining von syscalls durch umschalten des Modus (sysenter, sysleave). Der User-Mode verfügt dort über keine Schreibrechte.

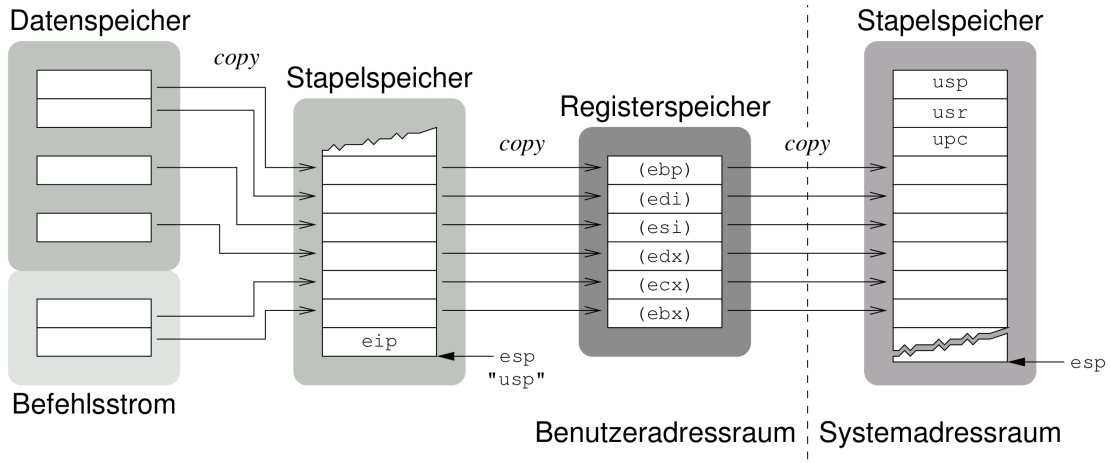


Abbildung 4: Primitivbefehl - call by value

Da die Operanden-Auswertung dynamisch zur Laufzeit erfolgt, müssen die Prozessorregister freigemacht werden. Solche primitiven Systemaufrufe sind häufig Unterprogramme. Kopieroperationen lassen sich einsparen durch fig. 5. Vergleich mit Komplexbefehl: fig. 6

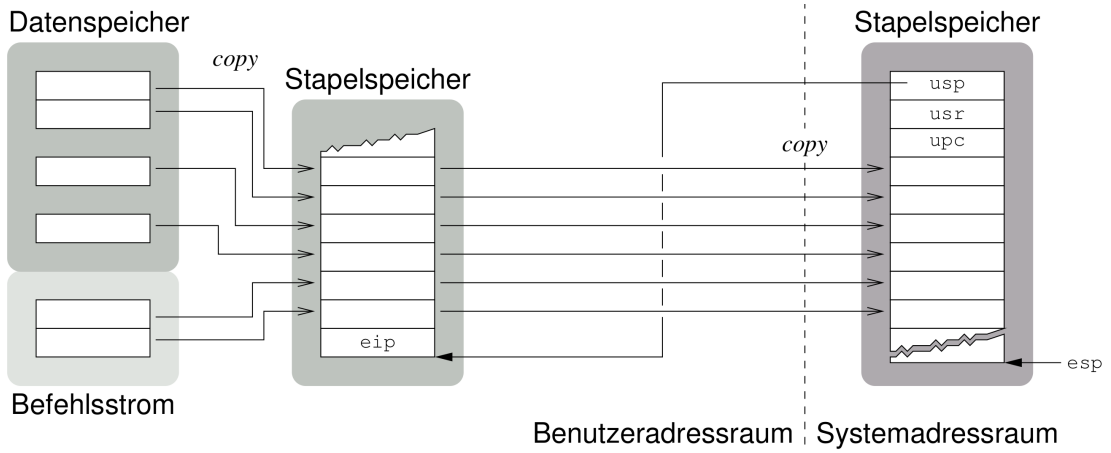


Abbildung 5: Primitivbefehl - call by value, ohne Umweg über Register
Das Betriebssystem lädt die Parameter direkt vom Userstack. Einsparung von Kopieraufwand möglich indem nicht x Parameter übergeben werden, sondern einer, der ein Pointer auf ein Parameter-Block ist. Der Kernel kann lesend und schreibend auf den Usermode-Stack zugreifen. Das ist genau das was ein Prozessor beim Abarbeiten eines Op-Codes tut. Die Adressierungsart gibt an wie viele Bytes vom Befehlsstrom zu lesen sind.

Allerdings hat der Prozess hierbei mehr Arbeit, dies entspricht dem RISC-Ansatz. Vergleich mit Komplexbefehl: fig. 6

Abschottung und bevorrechtigte Ausführung:

- durchzusetzende Eigenschaften:
  - priveligierter Arbeitsmodus für den Betriebssystemkern
  - Integrität - Verhinderung einer Sabotage ersterer Eigenschaft
- mögliches Mittel ist ein Trap, der vergleichsweise teuer ist (Zustandssicherung, Speicher- bzw. table look-up)
- Schneller geht das mit Spezialbefehlen (sysenter), weil hier kein Interrupt zugestellt werden muss und damit weder Suspendierung der aktuell laufenden Aktivitäten noch zwingend eine Kontextsicherung notwendig ist. Darüber hinaus ist keine Indirektion über Tabellen notwendig, sonder es wird das Kontrollregister MSR (model-specific register) verwendet. (s.u.)

Kontextwechsel der CPU ohne Kontext- und Tabellensuche - sysenter:

- sysenter
  - setzt CS, EIP, SS, ESP auf systemspezifische Werte

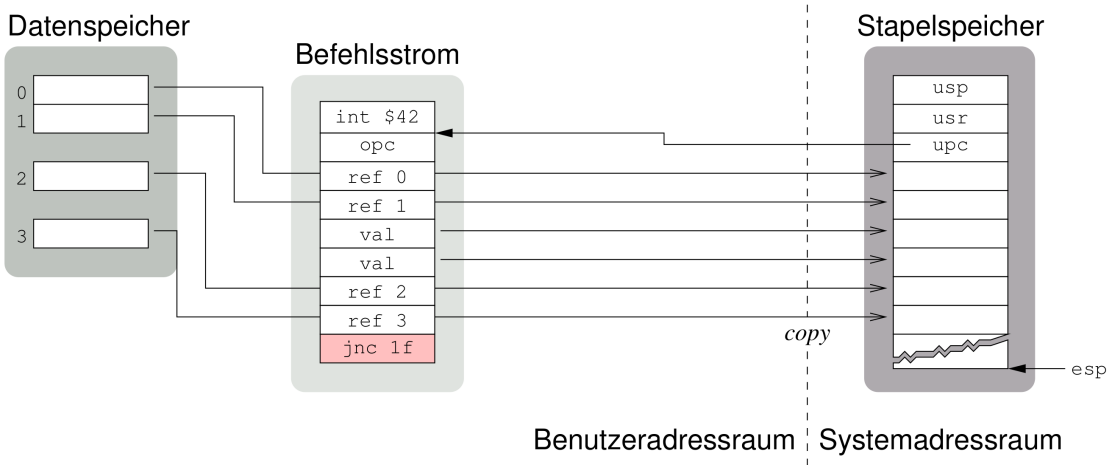


Abbildung 6: Komplexbefehl - call by value und reference - sowohl eine Werteübergabe (val) als auch eine Referenzübergabe (ref) ist möglich.
Hierbei hat der Prozessor mehr Arbeit und dies entspricht dem CISC-Ansatz. Für Direktwerte erfolgt die Parameterübergabe mittels call by value und ansonsten via call by reference. Solche komplexen Systemaufrufe sind häufig Makroanweisungen. Es wird hier also nur noch der Befehlsstrom und nicht mehr Befehlsstrom und Datenspeicher kopiert.

  - Deaktivierung der Segmentierung
  - Sperren von Interrupts
  - aktiviert Schutzring 0

- sysexit
  - setzt CS und SS auf prozessspezifische Werte
  - setzt EIP/ESP auf die in EDX/ECX stehenden Werte User-(Return address/-Stack Pointer) - diese müssen zuvor entweder von der Anwendung oder aber von dem C-Lib-Wrapper gesetzt werden, damit sysexit weiß wo es die Fortsetzung des Programms vorzunehmen hat.
  - aktiviert Schutzring 3 - nur von Ring 0 aus ausführbar

  Hierbei belegt das OS modellspezifische Register vor: MSR(model-specific register) CS - 174h, ESP - 175h, Eip - 176h, ...
  Also fungiert MSR[174h] als eine Art Basisindexregister in die Segmenttabelle und die Kontextsicherung liegt im Aufgabenbereich des Userprozesses.
  - bei sysenter: SS =MSR[174h]+8
  - bei syseexit: CS =MSR[174h]+16, SS = MSR[174h]+24

Prozessorregistersatz:

- SS,ESP sichern
- Statusregister und program counter sichern
- flüchtige und verwendete Arbeitsregister sichern, was sich mittels eines Stackwechsels zum Stackspeicher des Kerns umsetzen lässt.

Stapelspeicher:

- Dem Syscall einen Stack für den Kern zuteilen
- Bei abgeschottetem OS-Adressraum ist:
  - ein Stack im Kern für alle Fäden im Maschinenprogramm, typisch für ereignisbasierte Kerne (N:1)
  - ein Stack im Kern pro Faden im Maschinenprogramm, typisch für prozessbasierte Kerne (1:1)

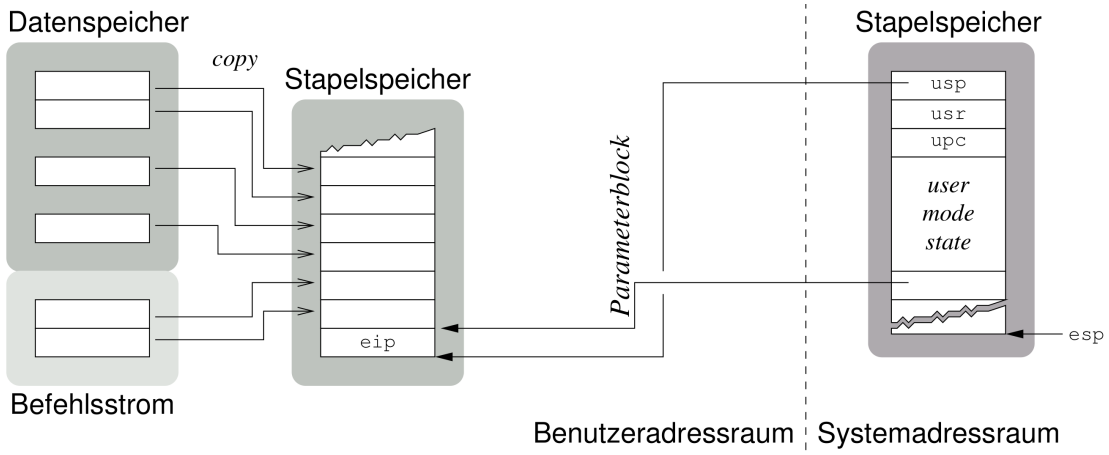


Abbildung 7: Primitivbefehl - call by reference
Die Systemfunktion lädt die Parameter direkt vom Userstack mittels indirekter Adressierung (Pointer auf den Parameterblock). Man verzichtet also auf die Ortstransparenz in der Systemfunktion und der Prozessorstatus ist betriebssystemseitig komplett gesichert.

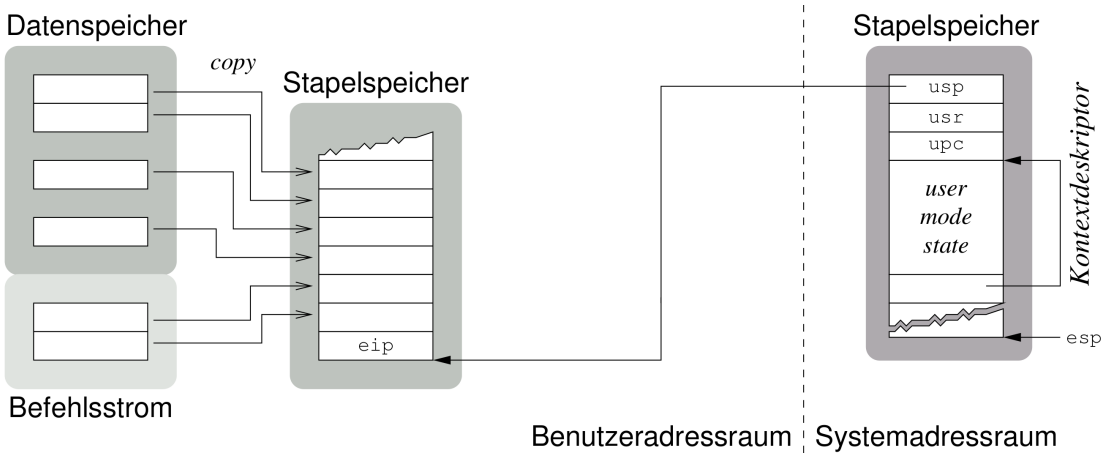


Abbildung 8: Primitivbefehl - call by reference mit Komplexdeskriptor
Systemaufrufparameter indirekt über einen Kontextdeskriptor laden, wobei der Parameterblock vom Userstackpointer abgeleitet wird. Dies unterstützt mehrkernbasierte Signalisierung von Fehlercodes. Dadurch wird der durch die CPU gesicherte Prozessorzustand offengelegt.

3 Betriebssystemarchitektur

Räumliche Trennung von Programmen/Prozessen:

- vertikal (vom Betriebssystem)
- horizontal (Anwendungsprogramme)

Rechnerarchitektur:

Attribute, die nichtfunktionale Eigenschaften in den Fokus stellen. Diese haben Auswirkungen auf technische Umsetzung eines Systems. Aus nichtfunktionalen Eigenschaften resultieren Zuverlässigkeit, Leistung, Effizienz, Safety, Security, Portierbarkeit, Aussehen, Benutzbarkeit, ...

- Operationsprinzip
  definiert funktionelles Verhalten durch Festlegung je einer
  - Informationsstruktur (Typen, Repräsentationen, anwendende Operationen)
  - Kontrollstruktur (Spezifikation, Interpretationsalgorithmen, Transformation der Informationskomponenten).
- Struktur
  Durch (HW-)Betriebsmittel Art und Anzahl und Kommunikationseinrichtung gegeben

Betriebssystemarchitektur:

Bestimmt durch ein Operationsprinzip für Systemsoftware und die Struktur ihres Aufbaus aus den einzelnen Systemprogrammen.

wdh. Monolith:

- ungetrennte Systemfunktionen - Architekturbildung durch Zusammenschluss aller Systemfunktionen
  - einheitlicher Adressraum
  - kleine Fehlereingrenzung
  - derselbe Arbeitsmodus: privilegiert
- bestenfalls schwache Modularität
  - hohe räumliche Verflechtung
  - unaufwändige Interaktion im System
  - prozedur-/prozessorientierter Aufbau

monolitisches schichtenstrukturiertes OS - fig. 9a:

Unterschiede zum normalen Monolithen: Die Interaktionen greifen nicht mehr über viele Ebenen hinweg miteinander, sondern sind klar strukturiert vertikal gekapselt.

- weiterhin ungetrennte Systemfunktionen - Architekturbildung durch Zusammenschluss aller Systemfunktionen
- starke logische Modularität, anstelle der schwachen
  - schwache räumliche Verflechtung
  - unaufwändige Interaktion im System
  - prozedur-/prozessorientierter Aufbau
- wohldefinierte innere Struktur
  - Programmhierarchie
  - Benutzhierarchie
  - funktionale Hierarchie

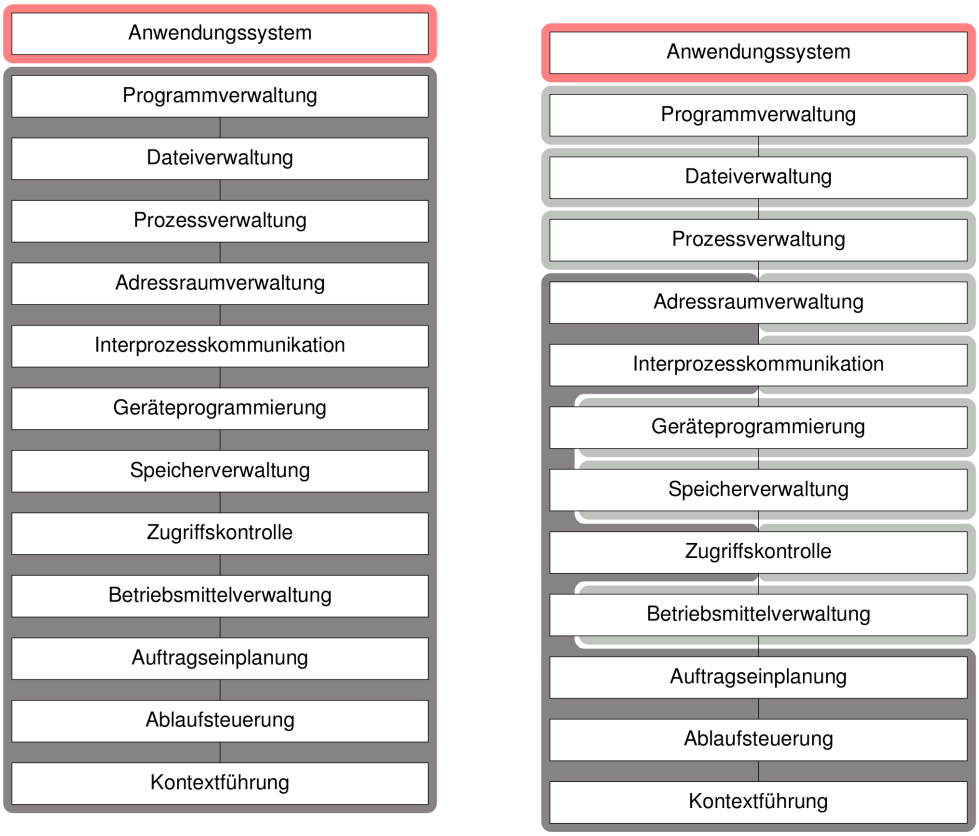
mikrokernbasiertes Betriebssystem - fig. 9b:

Auslagerung einzelner Systemfunktionen in separate Adressräume

- getrennte Systemfunktionen
  - in separierten Adressräumen
  - starke Fehlereingrenzung
  - verschiedene Arbeitsmodi (**privilegierter Kern,** unprivilegierte Systemprozesse)
- starke reale und logische Modularität
  - schwache räumliche Verflechtung
  - aufwändige Interaktion im System (**Kern - Systemaufruf,** Prozesse - IPC)
- mittelkörnige Struktur
  - Granularität auf Adressraumebene
  - Kernadressraum als Monolith

makrokernbasiertes Betriebssystem - fig. 10a:

- teils getrennte Systemfunktionen - Trennung zusammengeschlossener Systemfunktionen
  - in separierten Adressräumen
  - schwache Fehlereingrenzung
  - verschiedene Arbeitsmodi (**privilegierter Hybridkern,** unprivilegierte Systemprozesse)



(a) Monolithisches schichtenstrukturiertes OS

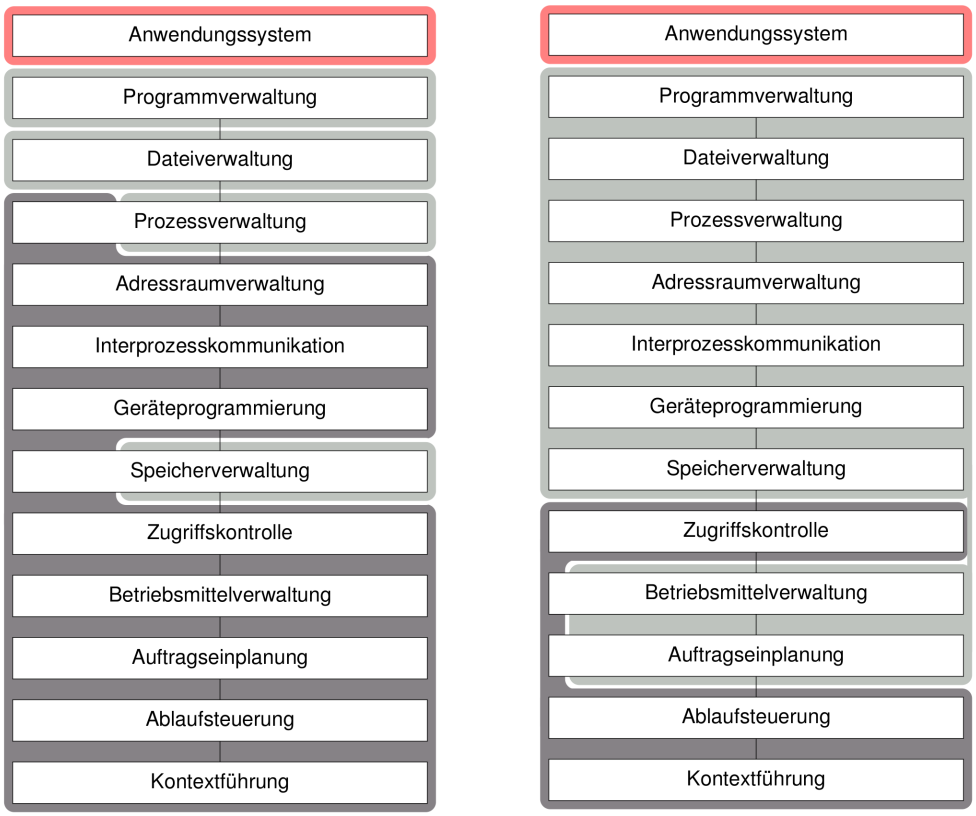
(b) Mikrokernbasiertes OS

Abbildung 9: Monolitisch und Mikrokern

- eingeschränkte Modularität
  - bedingte räumliche Verpflechtung
  - unaufwändige Interaktion im System (**Hybridkern - Systemaufruf, Prozesse -** IPC)
- grobkörnige Struktur
  - Granularität auf Adressraumebene
  - Hybridkernadressraum als Monolith

exokernbasiertes Betriebssystem - fig. 10b:

- Trennung von Belangen - Trennung von Verwaltung und Schutz
  - Betriebsmittelverwaltung - **Bibliotheksbetriebssysteme**
  - Betriebssystemschutz - **Exokern**



(a) Makrokernbasiertes OS (b) Exokernbasiertes OS

Abbildung 10: Makro- und Exokern

- konsequentes Freilegen von ...
  - Betriebsmittelbindungen
  - aller Betriebsmittelbelegungen
  - realer Betriebsmittelnamen
  - dem Widerruf von Betriebsmitteln
- die Grundtechniken
  - sichere Betriebsmittelbindung
  - sichtbarer Betriebsmittelwiderruf
  - imperativer Betriebsmitelentzug

Architekturformen:

- Nachrichten-Orientiert
  - Hier ist ein Empfangs-und Antwortprotokoll teil einer jeden Ressource
  - Realisierung mittels Monolitisch, mikrokernbasiert, makrokernbasiert möglich
  - kleine, recht statische Anzahl großer Prozesse
  - expliziter Satz von Nachrichtenkanälen zwischen Prozessen (meist langfristige Bindung)
  - relativ begrenzter Umfang direkt gemeinsam verwendeter Daten
  - Prozessidentifikation: Adressraum und statischer Kontext zu einem Prozess zugehörig, Schutzgrenzen werden selten übertreten, höchtens um Kurz den Kern zu betreten
- Prozedur-Orientiert
  - Hier ist ein Systemaufrufzuteiler zwischen den Prozessen und Ressourcen geschaltet
  - Realisierung mittels Monolitisch, exokernbasiert möglich
  - große, schnell ändernde Prozess-Anzahl mit kleinen Teilaufgaben
  - kurzfristige, schnelle Erzeugung und Beseitigung der Prozesse - Kommunikationskanäle müssen nicht aufgebaut werden
  - Kommunikation und Synchronisation über geteilte Daten - Schutz durch Zugriff über prozedurale Schnittstellen.
  - Funktionsidentifikation: Ausführungskontext ist einer sich in Ausführung befindlichen Funktion zugehörig
  - Systemressourcen werden als Datenstrukturen dargestellt - und sind global oder für mehrere Prozesse gemeinsam verwendbar
  - Anwendungen sind mit Prozessen assoziiert

Nachrichtenorientierung	Prozedurorientierung
sequentieller Prozess	ein-/wechselseitiger Ausschluss
Nachrichten (Kanal, Port)	Prozeduren (Name, Einstiegspunkt)
Anforderung (Antwort, Antwortzusage(future))	Aktivierung (Aufruf - Vorder-/Hintergrundausführung)
Empfang	Einsprung
Beantwortung	Rücksprung
Zuteilung (Selektionsanweisung)	Bindung (Prozedurreklaration)
selektiver Nachrichtenempfang	Bedingungsvariable

Tabelle 1: Gegenüberstellung Nachrichtenorientierung $\Leftrightarrow$ Prozedurorientierung

4 Hierarchien

Systemkomplexität:

- Programmhierarchie
  Ist vor Systemlaufzeit bedeutsam: statisch wenn Software konstruiert, entwickelt oder verändert wird. Die einzelnen Programme hinterlassen keine Spuren im System, wenn sie Makros sind.
- Prozesshierarchie
  Ist zur Laufzeit bedeutsam
- Mittelvergabehierarchie
- Schutzhierarchie

Struktur:
: partielle Beschreibung eines Systems

hierarchisch:
: Relation zwischen Teilpaaren/Ebenen

Hierarchische Struktur:
: Eine Struktur ist hierarchisch wenn bestimmte relationale Bedingungen gegeben sind s. 04-04 Solange die Art der Relation nicht mit angegeben wird ist diese Aussage recht inhaltslos. Bedingte Aufrufe müssen ausgeschlossen werden, da ein Programm sonst bei einer Hiararchiebildung nicht höher angeordnet werden, als die verwendete Maschine (Trap, Interrupt).

Hierarchie sequentieller Prozesse:
: Organisation von Systemaktivitäten über gleichzeitige, sequentielle Prozesse, um das System unempfänglich, in Bezug auf die Anzahl verfügbarer Prozessoren und deren Geschwindigkeit, zu machen. Betriebsmittel werden mittels Prozessen vergeben und auch Arbeitsaufträge und Informationen können unter einzelnen Prozessen getauscht werden.

  Von einzelnen Systemanforderungen wird nur eine endliche Anzahl Anforderungen an individuellen Prozessen zur Bearbeitung hervorgerufen und diese Anzahl ist relativ klein. Es genügt die Überprüfung eines jeden Prozess und dass jede Anforderungen an diesen immer nur eine endliche Zahl von Anforderungen hervorruft.

Hierarchie verwalteter Betriebsmittel:

Prozesse sind Eigentümer von Betriebsmitteln, können diese jedoch auch verteilen. Beispielsweise kontrolliert eine administrative Einheit die Zuteilung an die Prozesse. Mithilfe einer linearen Ordnung wird einer Zyklenentstehung vorgebeugt. Die Verwaltung eines Betriebsmittels kann auf verschiedenen Ebenen erfolgen (s. 04-04).

Eine Betriebsmittelvergabehierarchie ist eine Ergänzung der Programm-/Prozesshierarchien, um z.B. Verklemmungen vorzubeugen. Nachteilig kann eine schlechte Betriebsmittelauslastung durch schlechte Verteilung sein. Auch müssen Betriebsmittelanforderungen die Hierarchie durchlaufen, bevor sie zugelassen bzw. abgewiesen werden können. Beispiel:
1. Benutzerebene, 2. Systemebene, 3. Platzierung, 4. lokale Ersetzung, 5. globale Ersetzung

Hierarchie spezifischer Schutzzonen:

Ringförmige Anordnung der Schutzdomänen. Die unteren Ebenen haben uneingeschränkten Zugriff auf die höheren, andersrum jedoch nicht. Die Schutzhierarchie entspricht hier nicht der Programmhierarchie, denn Programmaufrufe können in beide Richtungen geschehen und Programme tieferer Ebene können von Programmen höherer Ebene profitieren. Dennoch wird sich die Schutzhierarchie häufig an der Programmhierarchie orientieren.

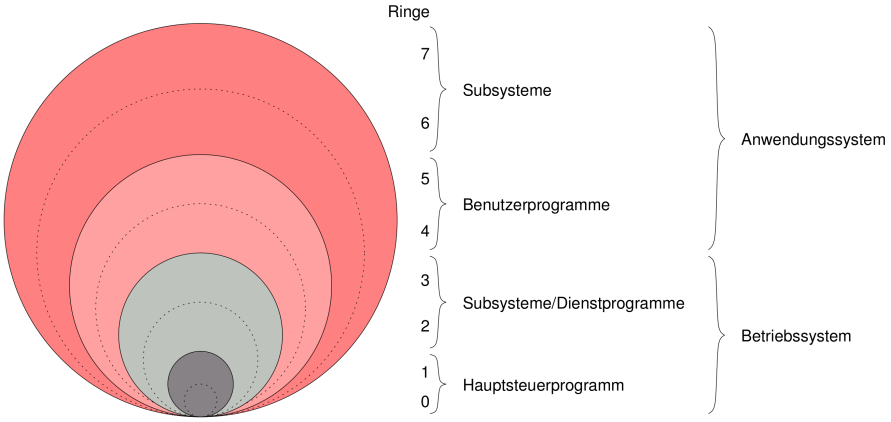


Abbildung 11: Schutzhierarchie - post Multics - auf Basis von Befähigungen

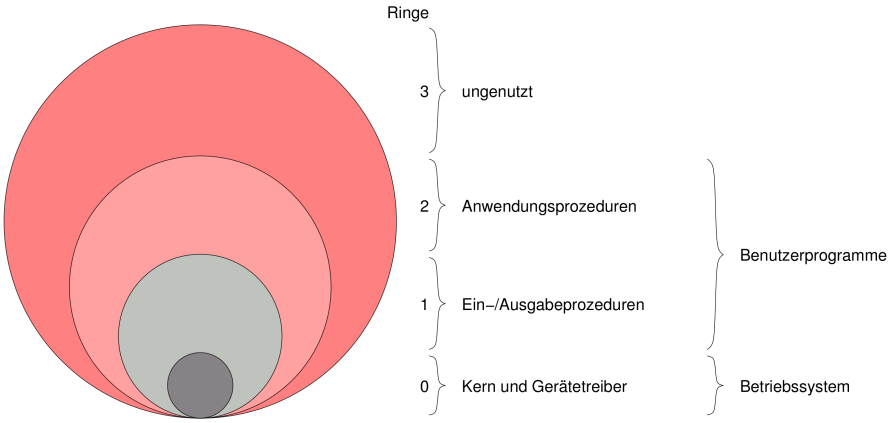


Abbildung 12: Schutzhierarchie - Verwendung - Durchgesetzt haben sich die Schutzhierarchien bisher nicht weitläufig, denn diese nachträglich in unstrukturierten Systemen einzubringen ist nicht trivial.

Benutzung von Programmen A, B:

A "benutzt" B, wenn B *korrekt* ausgeführt werden muss, damit A die eigene Arbeit entsprechend der Spezifikation vollenden kann. Weiterhin auch dann, wenn A nur dann korrekt funktionieren kann, wenn eine korrekte Implementierung von B existiert. "Benutzen" ist hier differenziert von "aufrufen" zu verstehen, denn:

- Programmaufrufe sind nicht zwingend eine Benutzung
  z.B: von A ist gefordert B bedingt aufzurufen, damit A die eigene Spezifikation erfüllt, sobald ein B-Aufruf generiert wurde, also auch wenn B falsch oder abwesend ist.
  Ein Korrektheitsnachweis für A muss lediglich Annahmen über die Art und Weise des Aufrufs an B machen.

- Es kann eine Benutzung stattfinden, obwohl kein Aufruf erfolgt, z.B. asynchrone Programmunterbrechung, Zuteilung realer Adressbereiche
  Die meisten Programme gehen davon aus, dass Unterbrechungsbehandlungsroutinen korrekt funktionieren. D.h. von Hardware ausgelöste Routinen terminieren obwohl ein Aufruf an sie in keinem Programm kodiert ist. Also sind Unterbrechungsbehandlungsroutinen die unterste Ebene einer Programmhierarchie. Darüber hinaus nutzen diese auch die Verwaltung des realen Adressraums.

Grundregeln zur Ebenenzuordnung:

- $Ebene_0$ umfasst die Programmmenge, die kein anderes Programm *benutzt*
- $Ebene_i, i > 0$ umfasst die Programmmenge, die mindestens ein Programm der $Ebene_{i-1}$ benutzt, jedoch keines auf höherer Ebene als $i - 1$

Entscheidungshilfe zu A "benutzt" B:

- A ist wesentlich einfacher, dadurch dass es B benutzt und B bereitgestellt wurde, um A zu unterstützen
- B wäre nicht wesentlich einfacher, wenn es A benutzte und B funktioniert ohne A, aber Kontextwissen in A könnte B effizienter ablaufen lassen
- Es gibt eine nutzbare Teilmenge, die B enthält, aber A nicht benötigt - A ist als eher eine Spezialisierung, wohingegen B eher einer Generalisierung entspricht.
- Es gibt keine Teilmenge, die A, aber nicht B enthält, denn dann würde A die von B bereitgestellten Funktionen nicht erfordern

Schicht einer funktionalen Hierarchie - Niveaumenge:

Fasst Programme derselben *Niveaumenge* der Benutztrelation zusammen. Also allen ist die gleiche *Abhängigkeitsebene* zugeordnet, alle *benutzen* den gleichen logischen Unterbau und alle werden von Oberbauten *benutzt*.

$Ebene_0$:

- Programme zur Unterbrechungsbehandlung
  Der first-level interrupt handler (FLIH) wird von allen *benutzt*, er muss also terminieren, damit der unterbrochene Prozess weiterlaufen kann und er hat zur Wahrung der Integrität den Prozessorzustand invariant zu halten
- Programme zur Verwaltung des realen Adressraums
  Die Zuordnung realer Adressen an Prozessinkarnationen wird ebenfalls von allen *benutzt*, Hauptspeicher muss vergeben, freigegeben oder entzogen werden und zwar korrekt, um systemweite Integrität zu wahren.

5 Adressraumverwaltung

Adressraumunterteilung:

Im logischen oder virtuellen Adressraum werden die Adressraumtypen mit Seite/Page benannt, im realen Adressraum hingegen als Seitenramen/page frame oder Kachel und wird stets errechnet aus einer Seitennummer und einem Offset. Eine typische Seitengröße ist 4096 Bytes.

Datenstruktur zur Adressabbildung:

Wird häufig von der Hardware (MMU) vorgegeben und umfasst typischerweise folgende Punkte, wobei die Seitendeskriptor-Struktur der Hardware unveränderlich ist

- Kachel-/Seitenrahmennummer (reale Adresse)
- Attribute
  - Schreibschutzbit
  - Präsenzbit
  - Referenzbit
  - Modifikationsbit
  - Privilegienstufe
  - Seiten(rahmen)größe
  - TLB
  - Weitere vom Betriebssystem definierte Attribute pro Seitendeskriptor werden häufig in Seiten-Kachel-Tabelle gehalten

```
1  struct ia32pd{
2     unsigned pd_present    : 1; // Seite im RAM, wenn true
3     unsigned pd_writable   : 1; // Schreibzugriff erlaubt, wenn
          true
4     unsigned pd_supervisor : 1; // Ring-3-Code darf auf die
          Seite zugreifen, wenn true
5     unsigned pd_through    : 1;
6     unsigned pd_uncached   : 1;
7     unsigned pd_referenced : 1;
8     unsigned pd_modified   : 1;
9     unsigned pd_index      : 1;
10    unsigned pd_global     : 1; // globale Speicherstelle
11    unsigned pd_avail      : 3;
12    unsigned pd_frame      : 20;
13 };
14
```

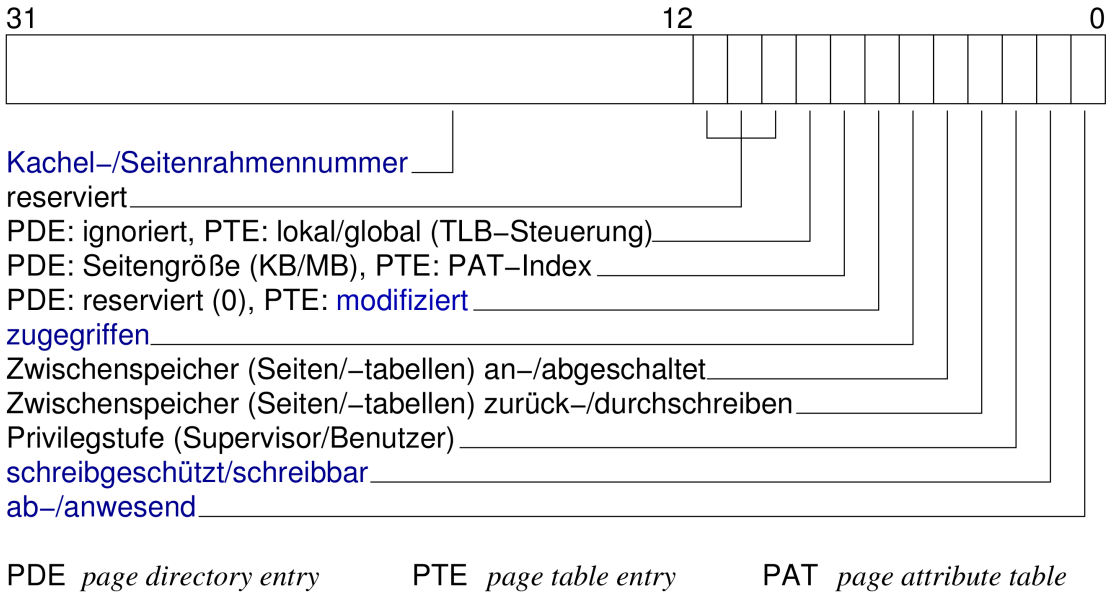


Abbildung 13: Seitendeskriptor

```
15 struct ia32pd_shadow{
16    time_t spd_loaded;
17    unsigned spd_count;
18 };
```

Listing 2: Seitendeskriptor - Bitfeld

Mögliche Ersetzungsstrategien: FIFO (Einlagerungszeit, Zeitstempel), LFU(Zugriffshäufigkeit, Zähler), LRU (second chance), ...
Pro Adressraum/Prozessinkarnation wird ein Tabellenpaar betrachtet. Bei Wiedereinlagerung sind einige Attribute zu beachten, deren Werte bei Auslagerung zu sichern sind, z.B. Bits 1 bis 11

```
1 //ra = real_address
2 //SKT = seiten_kachel_tabelle
3 //la = logical_address
4 //PSIZE = PAGE_SIZE
5 ra = (SKT[la/PSIZE].pd_frame * PSIZE) | (la % PSIZE)
```

Listing 3: einstufige seitenbasierte Adressumrechnung

Berechnungen einstufige seitenbasierte Adressumrechnung:

Swapping und Speichervirtualisierung ist etwas einfacher als mit Segmenten (Seitenrahmen haben alle die selbe Größe)
32-Bit-Maschine, 4KiB Seiten, 32-Bit Seitendeskriptor:
Tabellengröße $= \frac{2^{32}}{2^{12}} = 2^{20}$ Einträge, einer pro Seite/-ndeskriptor
Speicherplatzbedarf pro Prozessinkarnation (also meist logischen/virtuellen Adressraum):

Seitendeskriptorgröße: $\frac{32}{8} = 2^2 = 4$ Bytes
Tabellengröße: $2^{20} \cdot 2^2 = 2^{22} = 4$ MiB

Spätestens bei 64-Bit Maschinen ist das nicht mehr möglich, da viel zu viel Speicher notwendig ist (Petabyte-Bereich), weswegen man auch eher zu mehrstufigen Tabellen greift. Mehrstufigkeit bei non-Segmentierung mittels Tabellenhierarchien. Weil sonst viel zu Große Tabellen für MMU-Adressumrechnung notwendig sind (pro Adressraum).

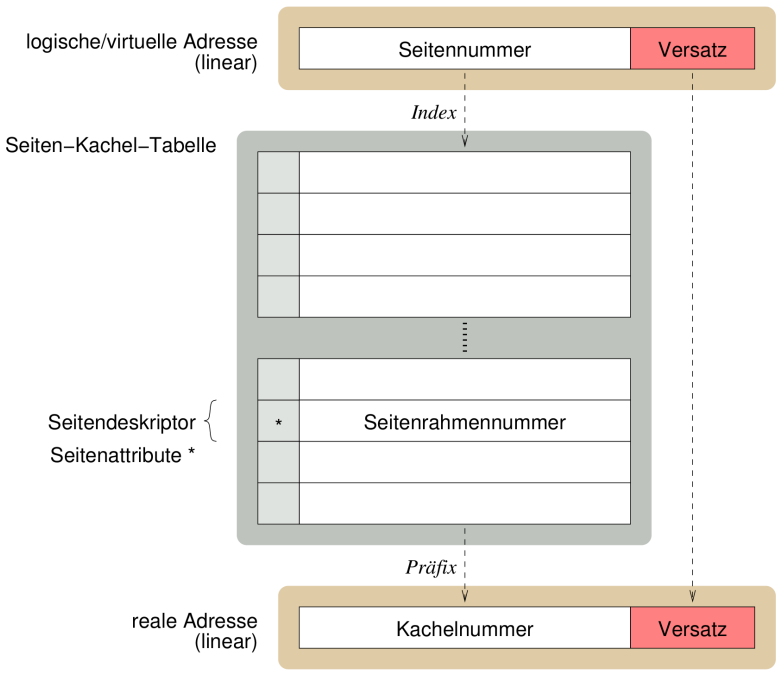


Abbildung 14: einstufig seitenbasierte Adressberechnung - entsprechende Berechnung: listing 3

```
1 //ra = real_address
2 //SKT = seiten_kachel_tabelle
3 //SVZ = seiten_verzeichnis
4 //la = logical_address
5 //PSIZE = PAGE_SIZE
```

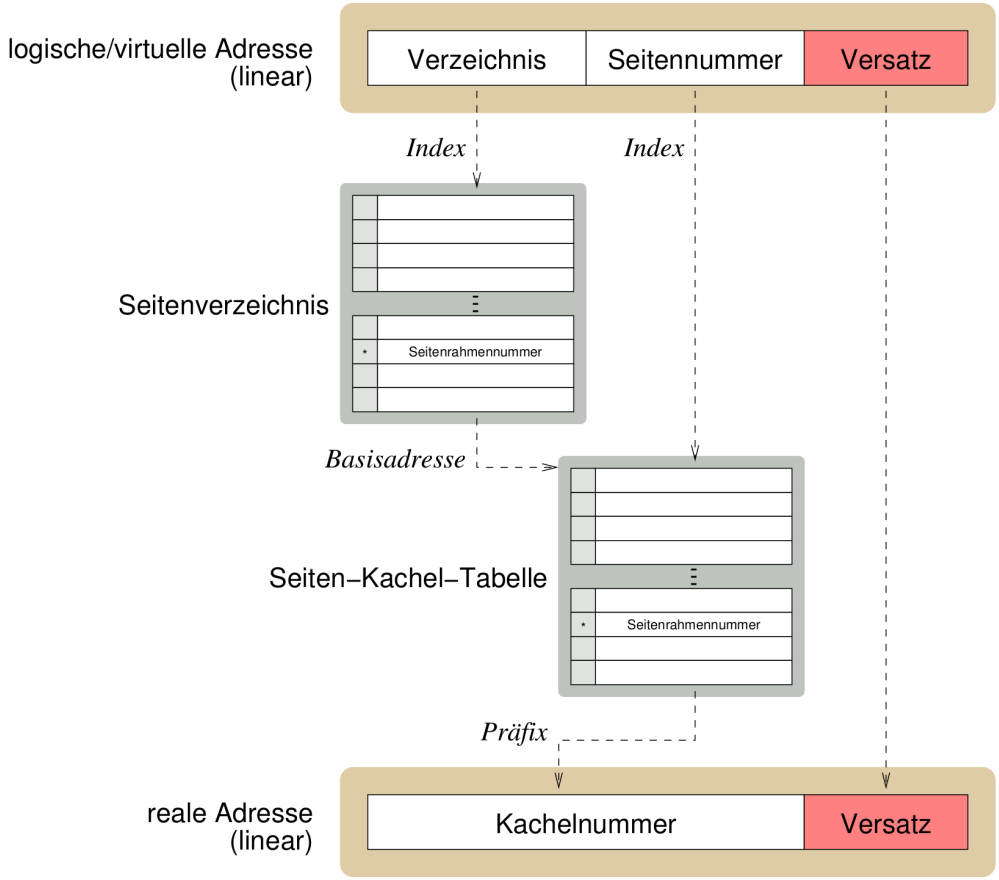


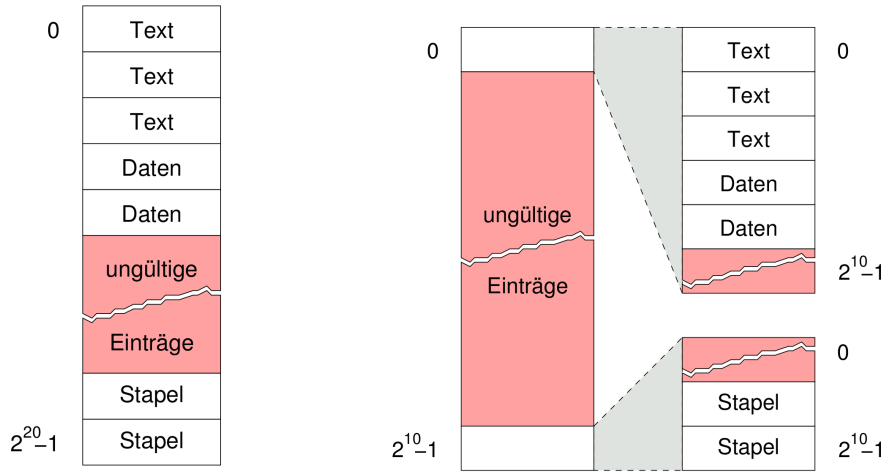
Abbildung 15: zweistufig seitenbasierte Adressberechnung - entsprechende Berechnung: listing 4
Text und Daten vs. Stack. Die Speicherverschwendung wird maßgeblich reduziert, da weniger ungültige Einträge.

```
6 //PTECOUNT = page_table_count
7 SKT = SVZ[la / (PTECOUNT * PSIZE)].pd_frame * PSIZE
8 ra = (SKT[(la % (PTECOUNT * PSIZE)) / PSIZE].pd_frame * PSIZE)
      | (la % PSIZE)
```

Listing 4: zweistufig seitenbasierte Adressberechnung - fig. 15

Berechnungen zweistufig seitenbasierte Adressumrechnung:

32-Bit-Maschine, 4KiB Seiten, 10-Bit Verzeichnis- und Seitennummer:
Verzeichnistabellengröße $= 2^{10} = 1024$ Einträge je 4 Bytes (ca.) $4KiB$
Seiten-Kachel-Tabellen-Größe: Berechnung identisch: $4KiB$
Also $2 * 4 = 8KiB$ pro $2^{10} \cdot 2^{12} = 2^{22} = 4MiB$ Adressraumabschnitt.



(a) einstufige Tabelle - $2^{20} - 7$ ungültige Einträge auf einer Tabelle

(b) zweistufige Tabelle - 3063 ungültige Einträge auf drei Tabellen

Abbildung 16: Anzahl ungültiger Einträge bei $12KiB$ Text-, $8KiB$ Daten- und $8KiB$ Stapelspeicher pro Prozess

Es gibt $1 < n \leq 2^{10}$ Seitentabellen pro Prozessinkarnation. Bei 64-Bit Adressen sind die Abbildungen bis zu 5-stufig. Für jeden Prozess gibt es mindestens drei Deskriptortabellen:

- Seitentabelle für Text- und Datensbereiche des Prozesses
- Seitentabelle für den Stapelspeicher des Prozesses
- Verzeichnistabelle mit je einem Eintrag für diese Seitentabellen

Invertierte Seitentabelle:

Ziel ist es den Platzverbrauch der einstufigen und mehrstufigen Seitentabellen zu reduzieren. Der Ansatz ist also anstatt für jede virtuelle Seite einen Eintrag vorzuhalten, dies nur für jede reale Seite zu tun. Zu einer virtuellen Adresse muss die reale dann in der Tabelle mittels einer Suche gefunden werden. Dies ist mit Hashtabellen kombinierbar. Die ausgelagerten Seiten einer invertierten Seitentabelle müssen bei einem Seitenfehler weiterhin anhand normaler Seitentabellen ausgelesen werden.

Eine Deskriptor-Tabelle spiegelt den realen, nicht logischen/virtuellen Adressraum wieder. Daher gibt es auch für jede Kachel/Seitenrahmen, und nicht für jede Seite, einen Deskriptor. Repräsentiert wird die Kachel-/Seitenrahmennummer mit einer Adressraumidentifikation eines geladenen Programms (aid) und einer Seitennummer (p).

Tabellenorganisation	Betriebssystem	Prozessor (MMU)
speicherschonenende und rechenintensive Repräsentation		
mehrstufig	komplex	komplex
gestreut invertiert	einfach	komplex
variabel linear invertiert	einfach	komplex
speicherintensive und bedingt rechenschonende Repräsentation		
einstufig	einfach	einfach
linear invertiert	einfach	komplex

Tabelle 2: Speicherschonende vs. Rechenschonende Repräsentation

Die Speicherabbildung sollte nicht linear für jeden Prozessorkern nur eine sein, da im realen Adressraum Lücken vorkommen, lieber eine individuelle Tabelle für jede Prozessinkarnation mit Tabellenumschaltung (TLB flushen), wobei die Tabellengröße von der Prozessadressraum-Größe abhängig ist.

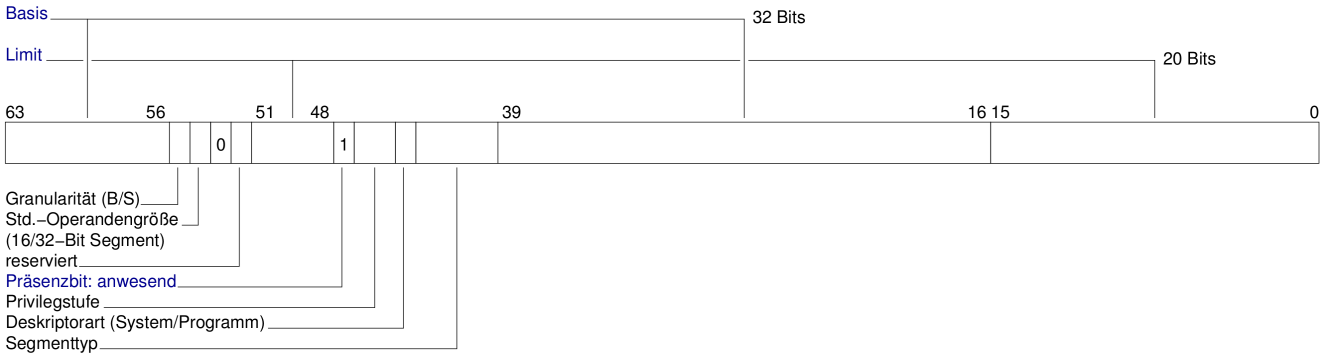
Verlinken, sowohl linear invertiert als auch gestreut invertiert

Segmentierung:

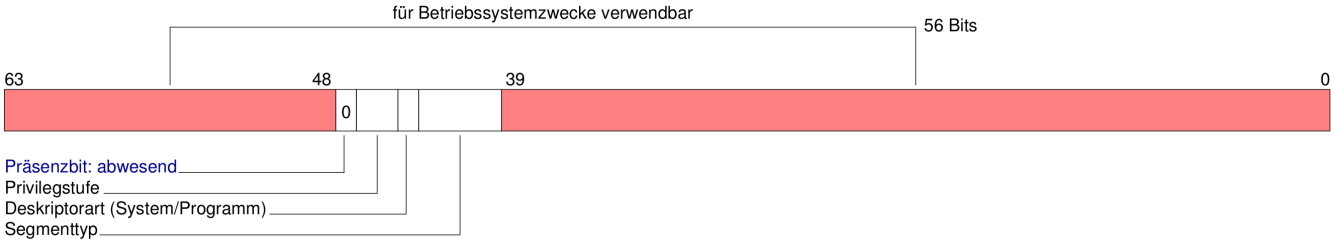
Segmente können unterschiedlich groß sein, werden jedoch intern durch gleichgroße Teile gebildet. Ein großer Vorteil ist die quasi-nicht-Existenz von ungültigen Adressen, diese werden bei der $<Limit$-Abfrage abgefangen. Wenn Segmentierung ohne Paging verwendet wird, so muss das gesamte Segment in den Speicher geswapt werden, wenn ein Zugriff auf diesen Bereich erfolgen soll.

Segmentdeskriptor-Informationen:

- Basis
  Segmentanfangsadresse im Arbeitsspeicher
  Ausrichtung entsprechend der Granulatgröße
- Limit
  Segmentlänge als Anzahl der Granulate
  Zahl der aufeinanderfolgenden Granulatadressen
- Attribute
  Typ (Text, Daten, Stapel)
  Zugriffsrechte
  Expansionsrichtung
  Präsenzbit
- Privilegstufe
- Klasse (interrupt, trap, task)
- Granulatgröße



(a) anwesend markiertes Segment



(b) abwesend markiertes Segment

```
1 //ra = real_address
2 //ST = seiten_tabelle
```

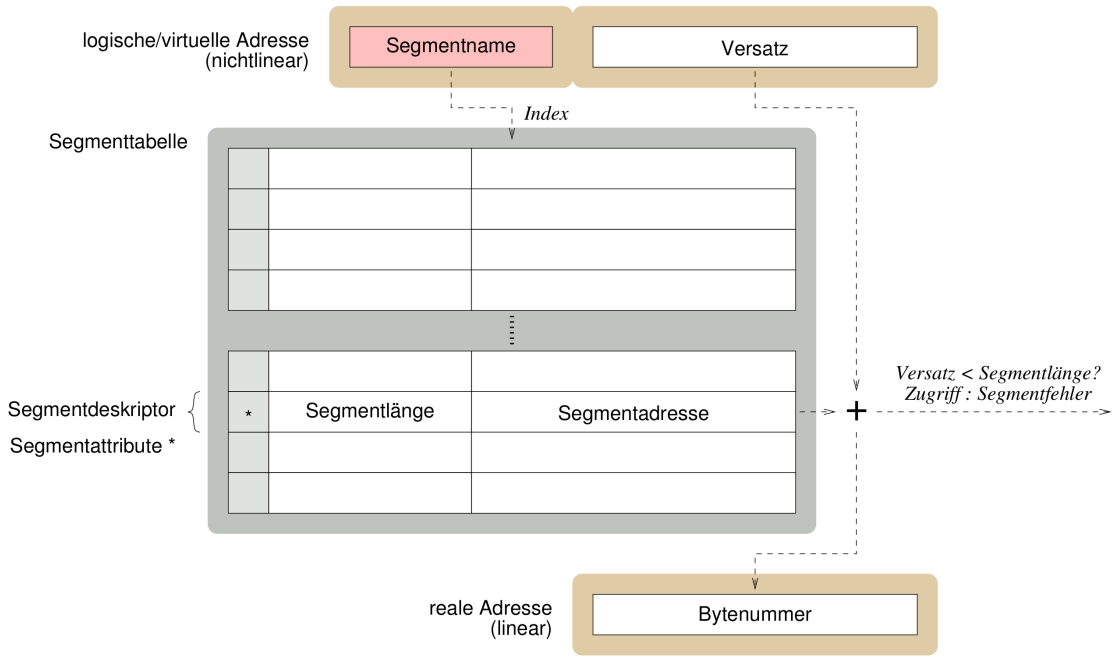


Abbildung 18: Segmentbasierte Adressierung - Berechnung: listing 5

```
3 //la = logical_address
4 ra = la < ST[sn].sd_limit ? ST[sn].sd_base + la : segmentation
      violation ↝ trap
```

Listing 5: Berechnung Segmentbasierte Adressierung - fig. 18

```
1 //ra = real_address
2 //SKT = seiten_kachel_tabelle
3 //la = logical_address
4 //PSIZE = PAGE_SIZE
5 SKT = (la / PSIZE) < ST[sn].sd_limit ? ST[sn].sd_base :
      segmentation violation ↝ trap
6 ra = (SKT[la / PSIZE].pd_frame * PSIZE) | (la % PSIZE)
```

Listing 6: Berechnung Segmentbasierte Adressierung - Seitennummeriert fig. 19

TLB:

Beim hardware-geführten TLB läuft die CPU/MMU die Tabellen ab, wodurch der Seitenfehler bei erfolgloser Tabellenwanderung verspätet kommt. Bei dem software-geführten TLB hingegen ist dies Teil einer Betriebssystemfunktion und der Fehler kommt unverzögert.

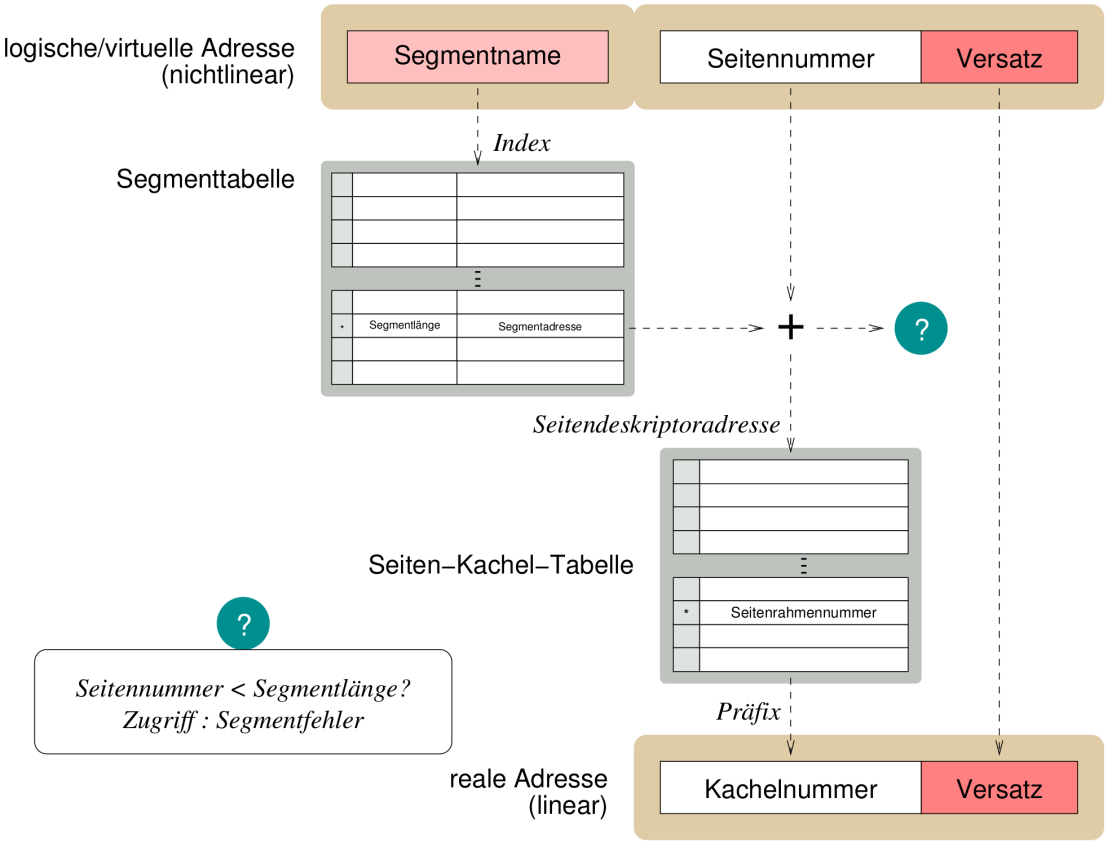


Abbildung 19: Segmentbasierte Adressierung - Seitennummeriert - Berechnung: listing 6

Gespeichert werden reale Adressen, nicht deren Inhalte. Gespeichert werden also lediglich die Inhalte der Seitendeskriptoren.

Ein TLB-Flush sorgt für Zugriffsfehler und ist bei kleineren TLBs relativ praktisch.

Einzelne Einträge zu taggen (beschildern) sorgt bei kleinen TLB zu Überläufen und darin resultierenden Performanceeinbrüchen. Verdrängt werden bevorzugt falsch beschilderte Einträge. Taggen von Adressraumindikatoren auf die Einträge der Seitentabelle ist praktisch, weil dadurch der Cache nicht geflusht werden muss, um das ”hit”von Adressen anderer Adressräume zu verhindern.

lustigerweise ist TLB gar nicht mal so viel langsamer als MMU

Puffereinträge taggen/beschildern:

- Adressraumbezeichner (address-space identifier, ASID)
  Ideitifkation der zu einem TLB-Eintrag zugeordneten Prozessinkarnationen
- Bereichsbezeichner (region identifier, RID)
  Generalisierung des ASID-Schemas, mehrere RID können aktiv sein.
- Schutzschlüssel (protection key, PK)
  ASID-Alternative - dienen nicht direkt der TLB-Eintragssuche. Assoziative Suche nach dem Schutzschlüsselregister
- Domänenbezeichner (domain identifier, DID)
  weniger Beschilderung, aber dennoch ähnlich zu einem Schutzschlüssel. Keine assoziative Suche, sondern Indizierte Domäneregister

6 Adressraummodelle

Mehradressraumkonzept:

Es gibt private Adressräume für Betriebssystem und Maschinenprogramme, wodurch gewisse Schutzdomänen gebildet werden. Dies ist möglich durch Vervielfachung eines realen Adressbereichs, entweder komplett, oder wie in $OOSTuBS_{BST}$ obere bzw. untere Teilbereiche.

- Vergrößerung verfügbarer Adressraum-Menge
- harte Speicherschutzgrenzen
- Aufräumen beim Ausstieg von Programmen einfach möglich
- Schwierige Kooperation zwischen Maschinenprogrammen, da Zeiger außerhalb der Grenze/Lebenszeit der Prozesse ungültig sein können und es schwierig ist Informationen zeigerbasiert auszutauschen, weshalb auf Kopieren zurückgegriffen wird.
- Mitbenutzung von Informationen nur über Umwege möglich, aufgrund der strikten Isolation der Anwendungskomponenten. Daher kann es auch keine shared library geben.
  Der Informationsaustausch kann erfolgen:
  - fensterbasiert, bedarfsorienterte Einblendung von Adressraumabschnitten - figs. 20 and 22
  - spezialbefehlbasiert, selektives Kopieren von Maschinenwörtern - fig. 21
  - adressraumgeteilt, direkter Zugriff von kompletten Benutzeradressraum
- implementieren (total/partiell) private Adressräume

Einadressraumkonzept:

Betriebssystem und Maschinenprogramme teilen sich einen Adressraum und der logische Adressraum bildet keine Schutzdomäne. Einzelne Adressen werden vom Betriebssystem erzeugt, verwaltet, zugeteilt, entzogen, zerstört. Prozesse greifen also auf sämtliche Objekte des Rechensystems zu.

Insbesondere durch ewig-gültige Adressen gefördert (breite Adressen)

- Adressierung und Schutz wird getrennt voneinander betrachtet
- profitieren von Prozessoren mit extrabreiten Adressen (64bit+)
- benutzen herkömmliche Adressraumverwaltungshardware

Virtualtisierung:

Virtualisierung des realen Adressbereichs, nicht des Hauptspeichers - entweder vollständig für das Betriebssystem, oder jeweils eine Teilmenge für alle Maschinenprogramme.

Bereiche der reale Adressen sind nicht physisch vorhanden, lediglich funktional, indem hinter jeder dieser Adressen eine speicherabbildbare Entität steht.

Private Adressräume:

Die MMU verhindert das Ausbrechen von einzelnen Prozessen aus den entsprechenden Adressbereichen. Informationsaustausch zwischen Programmen erfolgt mittels Maschinenbefehlen:

- Betriebssystem für die Maschinenprogramme (Ebene 3): Systemaufrufe zur Interprozesskommunikation oder Adressbereichsabbildung
- Zentraleinheit für die Betriebssystemprogramme (Ebene 2): privilegierte Befehle für Lese-/Schreibzugriff auf den Benutzeradressraum
- Pro
  - schränken verfügbare Adressbereiche nicht ein
  - geringe Anforderungen an das Betriebssystem
- Contra
  - Adressraumwechsel wird durch Systemaufrufe hervorgerufen
  - Vertikaler Informationsaustausch ist nur indirekt möglich

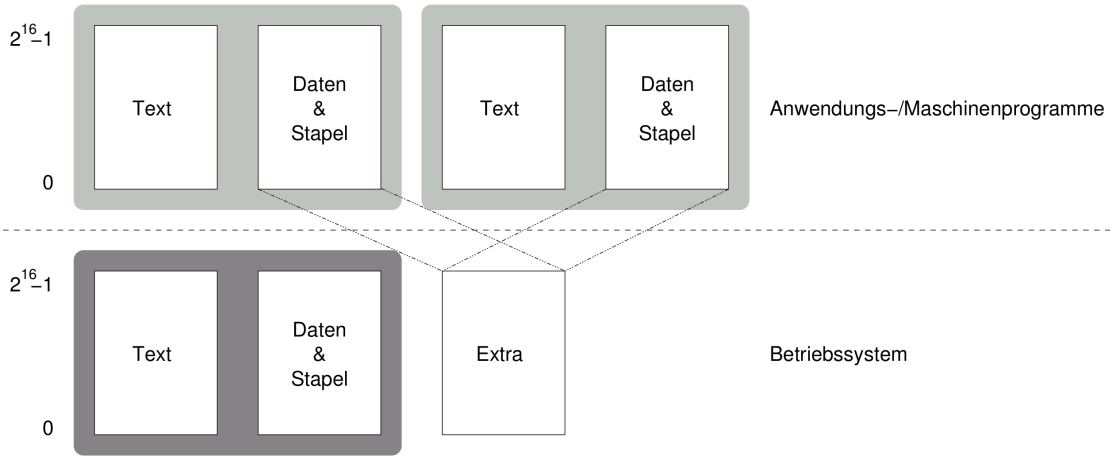


Abbildung 20: Fensterbasierter Ansatz - Intel 8086 (mikrokernbasiert)
horizontaler Informationsaustausch - Interprozesskommunikation(Nachrichten)
vertikaler Informationsaustausch - Zugriffe auf den Benutzeradressraum mittels eines Extrasegments

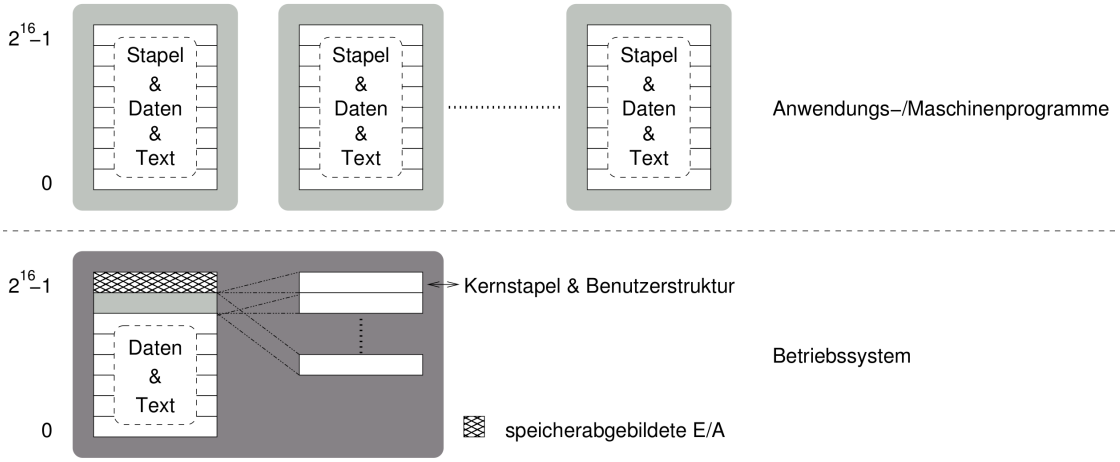


Abbildung 21: Spezialbefehlbasierter Ansatz - DEC PDP 11/40 (monolitisch)
horizontaler Informationsaustausch - Interprozesskommunikation (Nachrichten, Pipes), seitenbasierte Inkrement-Mitbenutzung
vertikaler Informationsaustausch - Zugriffe auf den Benutzeradressraum mittels Spezialbefehle, dies kann scheitern, sollten die Benutzeradressen ungültig sein, entsprechend muss der Kern hier Fehlerbehandlung vornehmen.

MMU-Rechte:

Bits 15-14 current mode
Bits 13-12 previous mode, vor der letzten Unterbrechung (trap, interrupt)

- 00 - kernel mode $\rightsquigarrow$ privilegiert, sicher, vertrauenswürdig
- 11 - user mode $\rightsquigarrow$ unprivilegiert, unsicher, zweifelhaft

Maßnahmen um mögliches Kernel-Scheitern zu vermeiden:

Ein Betriebssystem darf Benutzerprogramme nicht benutzen

- optimistisch
  - Annahme die für die Spezialbefehle verwendeten Daten sind gültige Benutzeradressen
  - Der Kern wird auf mögliche Unterbrechungen (trap) vorbereitet
  - Im Ausnahmefall findet die Fortsetzungsausführung im Kern statt
  - Vorgehen gword (pword)
    1. Sichern des Prozessor status Worts
    2. Sperren von Interrupts
    3. Sichern des Kernel detour pointers

4. Aufsetzen des detour für den Trap-Handler
5. (sende Daten die zu schreiben sind)
6. Daten auf den Kernel-Stack transferieren (vom Kernel-Stack transferieren)
7. Empfangen von Daten und vorbereiten für Rückgabe ()

- pesimistisch
  - Annahme die für die Spezialbefehle verwendeten Daten sind ungültige Benutzeradressen
  - Diese werden vor der Verwendung überprüft
  - Ausnahmefälle werden im Kern nicht zugelassen
  - Ist der Kern verdrängbar, ist dies anfällig für Race Conditions: Check der Adresse, Verdrängung des Kerns, Adressraumbelegung des Prozesses verändert sich, Adresse ist bei Prozesswiederaufnahme ungültig

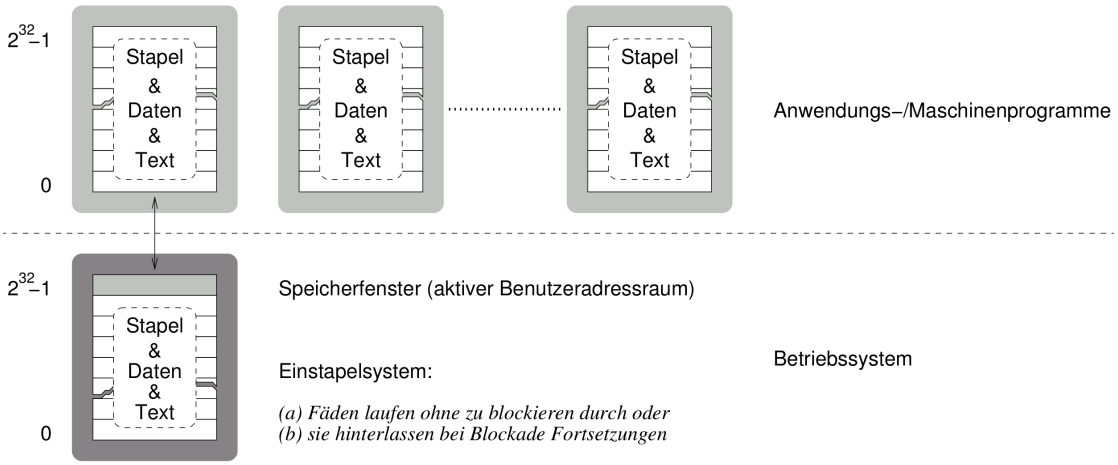


Abbildung 22: Fensterbasierter Ansatz - Darwin (mikrokernbasiert)
horizontaler Informationsaustausch - Interprozesskommunikation(Nachrichten), seitenbasierte Mitbenutzung
vertikaler Informationsaustausch - Zugriffe auf den Benutzeradressraum mittels eines Speicherfensters

Partiell private Adressräume:

Illusion eines eigenen physischen Adressraums für die Maschinenprogramme, Inklusion des Kerns. Der dem Betriebssystem total zugeordneten Adressbereich (A_t) existiert einfach, wohingegen die dem Maschinenprogramm partiell zugeordneten Bereiche (A_p) mehrfach, einmal für jedes Anwendungs- bzw. Maschinenprogramm, existiert. Es gilt $A_p \subset A_t$

Die MMU verhindert ein Ausbrechen von Prozessen aus A_p und A_t. Nicht jedoch ein Eindringen aus der Untermenge $A_t \setminus A_p$ hinein in A_p

Hauptaugenmerkmal ist weniger Adressraumwechsel hervorzurufen

- Pro
  - Systemaufrufe gehen ohne Adressraumwechsel
  - Vertikaler Informationsaustausch ist direkt möglich
- Contra
  - Verkleinerung des verfügbaren Adressbereichs
  - höhere Anforderungen an die Korrektheit des Betriebsystems

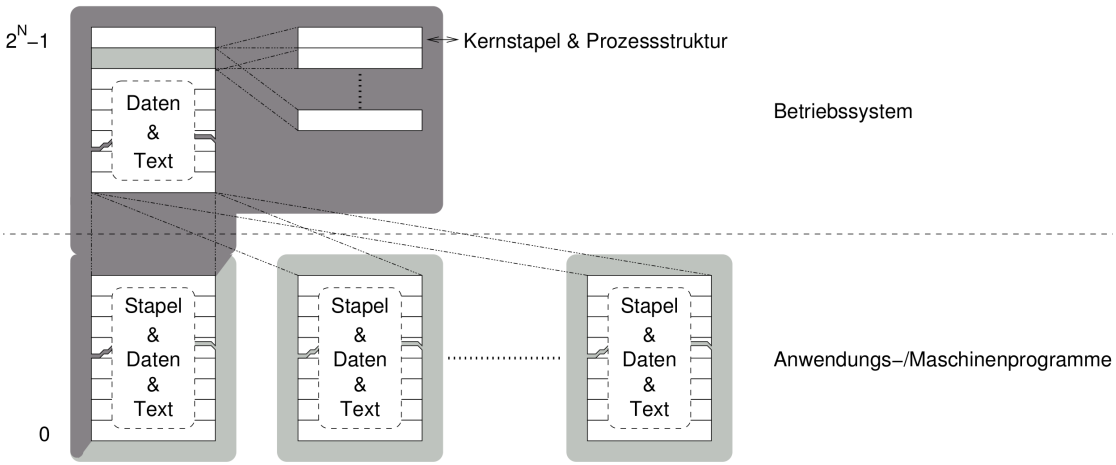


Abbildung 23: Inklusion des aktiven Benutzeradressraums
horizontaler Informationsaustausch - Interprozesskommunikation, Segment-/Seitenmitbenutzung
vertikaler Informationsaustausch - speicherabgebildeter Zugriff auf den Benutzeradressraum

Teilung des Adressbereichs:

Die Aufteilung kann unter Benutzer und Betriebssystemadressraum gleichmäßig oder ungleichmäßig aufgeteilt werden.

Prinzip des single-address-space operating system (SASOS):

Trennung von Belangen (separation of concerns). Verwendet gerne kennwortgeschützte Befähigung und ist frei kopier-, speicher- und kommunizierbar und für dessen Verwendung ist ein Einbezug des Betriebssystems nicht notwendig

- Adressierung
  - Adressen sind eindeutig und potentiell für immer gültig
  - Adressen sind kontextunabhängig
- Schutz
  - Nicht jedes Datum ist von einem Programmfaden aus zugreifbar
  - Schutzdomäne definiert die Zugriffsrechte von Fäden
  - Zugriffsrechte ändern sich beim Wandern durch Schutzdomänen

Selbst bei einer vergleichsweise kleinen realen Adressbereite können virtuell breitere Adressen verwendet werden. Bei Sicherung der Zugriffsrechte von Dateien/Objekten (Segmente, Geräte, Maschinenbefehle, ...) muss garantiert werden, dass die Rechtezuordnung nicht manipuliert werden kann. Dies kann beispielsweise anhand von Listen (capability list) oder Kennwörtern, die vom Objektverwalter bei dem Deskriptor vergeben werden, geschützt werden. Bei Hardwarezugriffsrechten (lesen, schreiben, zugreifen, ausführen) werden Deskriptortabellen verwendet.

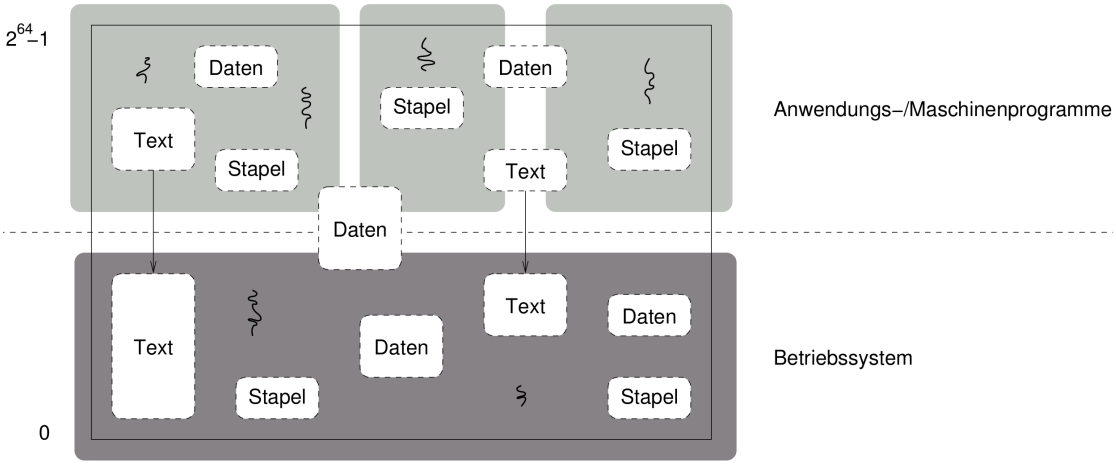


Abbildung 24: Hyperadressraum mit Schutzdomänen (mikrokernbasiert)
horizontaler Informationsaustausch - Unterprogrammaufrufe, gemeinsame Adressbereiche
vertikaler Informationsaustausch - genauso

7 Sprachbasierung

Sicherheitseigenschaften:
Schutz in räumlicher und zeitlicher Hinsicht.

- Immunität *(security)*
  Angriffssicherheit, Schutzt einer Identität vor seiner Umgebung, verhindern in einen Adressraum eindringen zu können
- Isolation *(safety)*
  Betriebssicherheit, Schutz der Umgebung vor einer Entität, verhindern eines Ausbruchs aus einem Adressraum

Isolation durch Typsicherheit und Gefährdung durch Schwachstellen:
typsichere Programmiersprachen und Compiler, sowohl statisch typisierte, als auch dynamisch typisierte Sprachen. Zeiger als Datentypen sind hierbei problematisch. Problematisch sind jedoch nicht nur Zeiger:

- Feld - Über-/Unterschreitung von Feldgrenzen
- Zeiger - Wertezuweisungen an/Änderungen von Zeigervariablen
- Rekursion - Laufzeitstapel unberechenbar ausdehnbar
- Argumente - Übergabe beliebiger Menge von Parametern
- Typisierung - casts
- Außenreferenz - Aufrufen eines (externen) Unterprogramms

Der Speicherschutz wird häufig nicht hardwarebasiert gewährleistet, sondern muss mittels typunsicheren Sprachen implementiert werden, was bei typsicheren Betriebssystemen sonderbar wirkt, aber es ist nun mal eine echte Systemprogrammiersprache erforderlich.

Sichere Programmiersprache:
Ist frei von oberen Schwachstellen oder der Compiler bietet Möglichkeiten der Absicherung.

Modul:
Vereint Programme, die einen Hauptschritt in der Verarbeitung ausmachen oder dem Prinzip des information hidings folgen. Letzteres wird häufig als *das* Merkmal von Modulen gehandelt. Modularität wird nicht durch Schutzmaßnahmen des Betriebssystems erreicht.

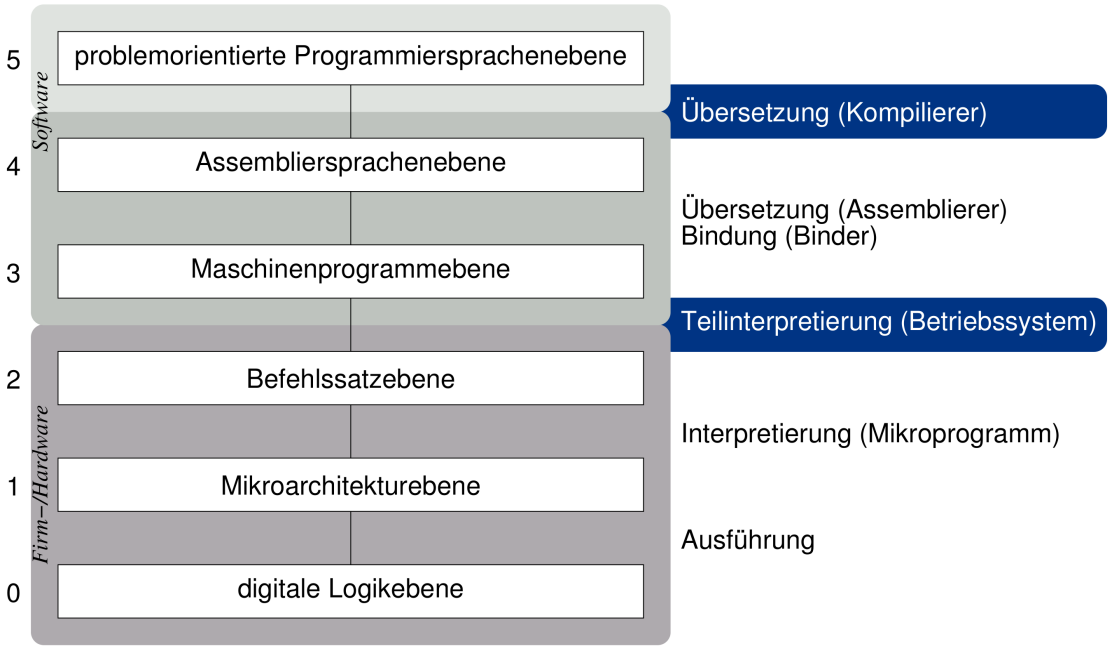


Abbildung 25: Trennung von Belangen - separation of concerns - Körnigkeit

Modularisierung liegt vor, wenn Module entwickelt werden können, ohne viel Wissen über innere Strukturen anderer Module zu haben und Austausch oder Refactoring eines Moduls möglich ist, ohne andere Teile des Systems anfassen zu müssen.

Granularität und Schutz:

Grobkörniger Schutz des Speichers (Kachel-/Segmentweise Speichververwaltung, ungepufferte I/O, Prozesse) sollte auf Betriebssystemebene erfolgen, wohingegen die feinkörnige Kontrolle auf Sprachebene von Compilern und den Laufzeitsystemen bereitgestellt werden sollten (dynamische Speicherverwaltung, gepufferte I/O, Programmfäden ...).

Hard protection boundaries:

Nichtsdestotrotz sind harte Sicherheitsgrenzen notwendig, um das System gegen zweifelhafte Parteien oder unsichere Sprachen abzusichern. Dies ist jedoch nur richtig, wenn das Rechensystem ein Mehrsprachensystem mit verschiedenen Arten von Dialogbetrieben und ein general-purpose computer ist

Systemimplementierungssprache:

- Flansch/flange
  - Unterbrechungsbehandlung wird ummantelt (erst flüchte Register sichern und anschließend vor dem Rücksprung deren Wiederherstellung)
  - Zugriff auf gestapelten Prozessorstatus (Trap)
  - kann sowohl synchron (Vektornummer 128 mit Systemaufruf und Wertübergabe des Prozessorstatus) als auch asynchron (Vektornummer 42 unter Angabe der Signalisierungsart, da Unterbrechungen gesperrt bleiben müssen) - letzteres erfolgt eher mit Templates, statt mit Parameterübergabe
  - Vorgehen flange
    1. sichern der volatile (flüchtigen) Register
    2. ausführen der first-level Exception-Handler
    3. wiederherstellen der volatile (flüchtigen) Register Rückkehr vom Trap/Interrupt
- Koroutine
  - Grundlage für Programmfäden/Prozessinkarnationen
  - Ausprägung für ereigns- und prozessbasierte Systeme
- Prozessorstatus
  - kontextabhängiger Maschinenzustand einer Koroutine
- Transaktion
  - echte Elementaroperationen: TAS, FAA, CAS, ...
- Spezialbefehle
  - Unterbrechungssteuerung LL/SC
  - TLB (ÜBersetzungspuffer) laden/flushen
  - asynchrone Systemsprung-Auslösung (AST)
  - Ruhezustand, Bereitschaftsbetrieb
  - Speichersynchronisation
  - Prozessorkernsignalisierung
- Maschinenwort
  - Repräsentation des Speicherworts des Prozessors
- Speicherfeld
  - Repräsentation des realen Adressraums (Tabelle)

Kontrollfluswechsel:

- ereignisbasierte Systeme (gemeinsamer Stack)
  1. Behalte Ziel-Instruktions-Pointer
  2. Rückgabeadresse dieser Koroutine
  3. Umschalten zur nächsten Koroutine
- prozessbasierte Systeme (eigenständige Stack)
  1. Behalte Ziel-Stack-Pointer
  2. Erzeuge Adresse dieser Koroutine
  3. Rückgabe des Stack-Pointers
  4. Schalte zu dem Stack der nächsten Koroutine um und sorge für dessen Fortführung

Kontrollflusserzeugung:

- ereignisbasierte Systeme und prozessbasierte Systeme
  1. Kind-Start-Adresse
  2. Signalisiere Fortführung von Elter
  3. Initialisiere Fortführung des Kindes
  4. Stelle Adresse wieder her und überprüfe diese

Sicherung/Wiederherstellung des Prozessorstatus:

Hinsichtlich des aktuellen Maschinenzustands des Prozessors kontextabhänigge Koroutinenwechsel, wobei nur aktive Prozessorregister beachtent werden müssen, weshalb der zu sichernde Sicherungspuffer von variabler Größe ist, maximal jedoch alle definierten Register.

Arbeiten auf Behältnis statischer Größe:
dump(bin) - in den Sicherungspuffer abladen
pick(bin) - aus dem Sicherungspuffer aufsammeln

sti muss oder darf nicht gesetzt werden bei der Interruptbehandlung, je nachdem ob die Signalisierung level oder edge-Triggered ist. Bei level-triggered dürften die Interrupts nicht so häufig kommen, dass der Stack überläuft, da sonst panic.

Ziemlich cool auf Seite 25 ist dann der Code in C++, der zuvor in Assembler geschrieben werden musste. Das macht die Sache relativ komfortabel.

8 Interprozesskommunikation

Gleichberechtigter Nachrichtenaustausch - fig. 26:

Ein Prozess ist Sender, der andere Empfänger, jedoch können die Prozesse diese Rollenverteilung tauschen, was vorsichtig zu erfolgen hat, da die Rollen zum Kommunikationszeitpunkt verschieden sein müssen, um Verklemmung vorzubeugen. Der Sender benötigt ausschließlich wiederverwendbare, wohingegen der Empfänger darüber hinaus noch konsumierbare Betriebsmittel benötigt.

- send: versendet eine Nachricht an einen Empfänger, erwartet den Nachrichtenausgang, -eingang oder -empfang
- receive: empfängt eine Nachricht vom Sendeprozess, erwartet den Nachrichteneingang

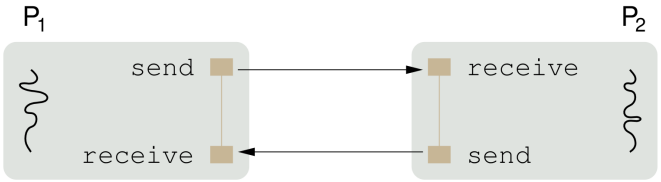


Abbildung 26: Gleichberechtigter Nachrichtenaustausch

Ungleichberechtigter Nachrichtenaustausch - fig. 27:

Eine Art Hierarchie, also P2 nimmt beide Rollen an, jeweils für den anderen Gesprächspartner.

- send: versendet einen Auftrag an einen Empfangsprozess und erwartet die Auftragsbeantwortung
  darf theoretisch blockieren (puffer voll)
- receive: Erwartet/empfängt einen Auftrag von einem Sendeprozess
- reply: versendet eine Antwort an den beauftragenden Sendeprozess
  sollte besser nicht blockieren, deswegen ist das, gleichwohl möglich, problematisch send-receive-reply durch send-receive nachzubilden (fig. 28)

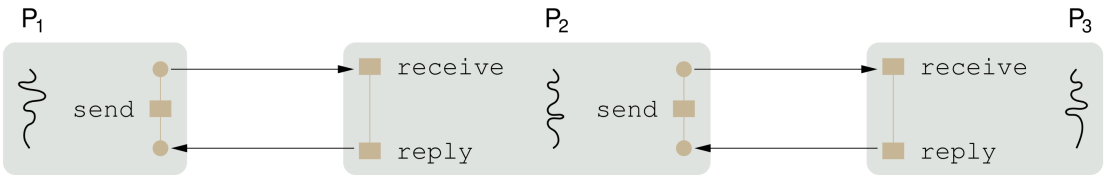


Abbildung 27: Ungleichberechtigter Nachrichtenaustausch

Nachbildung send-receive-reply durch send-receive - fig. 28:

Diese Nachbildung ist problematisch, da reply besser nicht blockieren sollte. Der Auftragnehmer P_2 muss dem Auftraggeber P_1 vertrauen, dass keine Blockade hervorgerufen wird. Der Systemprozess soll also einem Anwendungsprozess bzgl. gutartiger Verwendung vertrauen und der Systemprozess benutzt den Anwendungsprozess. Entsprechend ist das konzeptionell kaputt, weil der Server einem Benutzer nicht vertrauen müssen sollte

- reply: blockiert nicht, da Auftraggeber P_1 die Antwort erwartet, P_1 hat die Betriebsmittel (Puffer) für P_2 garantiert
- send: lässt P_2 beim Versenden der Antwort blockieren, wenn P_1 noch nicht empfangsbereit oder das Betriebsmittel (Puffer) zum Empfang noch nicht garantiert hat.

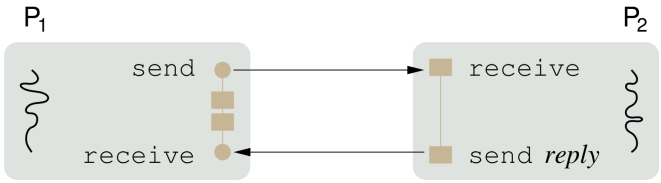


Abbildung 28: Nachbildung send-receive-reply durch send-receive

Wichtig ist erst zu blockieren, bevor man sendet, sonst ist lost-wakeup möglich, weil send, receive nicht atomar sind. Fehlende Verfügbarkeit von Puffern für Antworten sind öfters ein Problem, da Senden blockieren kann.

Aktionsunterscheidung:

- Datentransfer
  ähnlich einem Speicherdirekzugriff (DMA), reale Übertragung gespeicherter Informationen, über Prozess-, Adressraum-, Kern- oder Rechnergrenzen hinweg
- Synchronisation
  Fortschritt des Empfangsprozesses hängt vom Sendeprozess ab - konsumierbar - Vorhalten der Nachrichten im Puffer.
  Fortschritt des Sendeprozesses hängt vom Empfangsprozess ab - wiederverwendbar - Pufferplatz

Prozessinteraktion:

erfolgt entweder über Nachrichten (message passing) oder aber über Speicherdirektzugriff (DMA)

- message passing - über gemeinsames Betriebsmittel
  - synchron und blockierend
    - Sender wartet passiv im *send* auf das *receive* des Empfängers
    - Empfänger wartet passiv im *receive* auf das *send* des Senders
    - Datentransfer erfolgt, wenn beide über diesen übereinstimmen.
  - asynchron und blockierend oder nichtblockierend
    - ausnahmebedingtes Warten im *send* oder *receive* bei vollen Puffern, leeren Nachrichten
    - Um die asynchronität abzubilden, werden womöglich wiederverwendbare Betriebsmittel benötigt, weshalb blockieren weiterhin möglich ist. Sonst müsste man das Scheitern von send/receive zulassen.
- DMA

Aktive Nachricht:

Enthält im Header die Adresse der Empfangsroutine, die den Empfang der restlichen Nachrichten vornehmen soll (Identifikation). Dies läuft verdrängungsfrei (run to completion). Dieses Modell ist vergleichbar mit einer Unterbrechungsbehandlung, die unmittelbar im Anwendungskontext erfolgt. Zugrunde liegt das Programmiermodell dem SPMD (single program, multiple data), also alle Prozessoren führen dasselbe Programm aus entweder mit shared oder mit distributed memory innerhalb des selben logischen Adressbereichs. Jeder Prozessor operiert jedoch auf eigenen Datensätzen.

Ablauf bei Verarbeitung einer aktiven Nachricht:

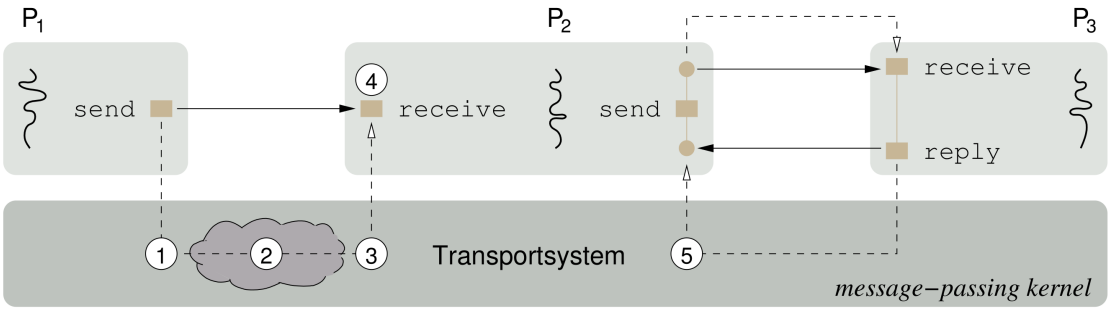
1. Unterbrechung der laufenden Berechnung (Ankunft des Nachrichtenkopfes)
2. Auslösung der ersten Unterbrechungsbehandlungsstufe (Sicherung)
3. Aufruf der Nachrichtenbehandlungsroutine (Kontexterzeugung)
4. Behandlung der aktiven Nachricht (Extraktion/Integration)
5. Beendigung der Behandlungsmaßnahmen (Kontextzerstörung)

send-Varianten - fig. 29a:

- no-wait send, non-blocking send
  - schwierig, wenn eine hohe Wahrscheinlichkeit des erfolgreichen Sendens garantiert werden soll
  - Sendeprozess wartet, bis Nachricht komplett erzeugt wurde
- blocking send
  - Sendeprozess wartet bis Nachricht sendeseitig freigestellt wurde (in das Transportsystem eingespeist wurde)
- reliable-blocking send
  - Sendeprozess wartet bis die Nachricht auf Empfangsseite postiert wurde (in einem Nachrichtenpuffer gespeichert und dem Empfangsprozess zum Empfang zugeordnet wurde)
- synchronization send, explicit-blocking send
  - Erhalten einer Nachricht, impliziert nicht dessen Abarbeitung
  - Sendeprozess wartet bis die Nachricht vom Empfangsprozess entgegen genommen wurde
- remote-invocation send, request/reply
  - send-receive-reply, also Bestätigung nach Abarbeitung
  - Sendeprozess wartet bis Nachricht vom Empfangsprozess entgegengenommen, verarbeitet und beantwortet wurde

Kommunikationsendpunkte verschiedener Abstraktionsebenen - fig. 30:

- Prozesskontrollblock
  - direkte Adresse des Sende-/Empfangsprozesses
  - Deskriptor einer Prozessinkarnation



(a) Verschiedene Sende-Varianten im Vergleich

Zugriff auf einen noch nicht versendeten Block führt trotzdem zum page fault, weil der Zugriff dadrauf nicht mehr erwünscht ist, da bereits ein Kopier-Vorgang in das Transportsystems stattfand (blocking-send)

1. non-blocking send (no-wait send) - Nachricht ist evtl. noch da
2. blocking send - Nachricht ist auf dem Weg (im Netzwerk)
3. reliable-blocking send - Nachricht ist im Puffer angekommen
4. explicit-blocking send (synchronisation send) - Nachricht angenommen und vom Prozess aus dem Puffer entnommen
5. remote-invocation send (request/reply) - Nachricht wurde verarbeitet

- Prozessanschluss
  - indirekte Adresse des Sende-/Empfangsprozesses
  - Zugang (Port) zu einer Prozessinkarnation
- Postkasten
  - Adresse eines Behältnisses für Nachrichten
  - Aus- oder Eingang (mailbox)
- Prozedur
  - Adresse der Verarbeitungsroutine der Nachricht
  - Nachrichtenextraktion/-integration

Eindeutigkeit der Kommunikationsendpunkte:

Kommunikationsendpunkte müssen eindeutig sein. Möglichkeiten hierfür sind *Zufallszahlen, Zeitstempel* (erfordert synchronisierte Uhr), *Knotennummer* (MAC-Adresse), Generationszähler, reale Adressen.

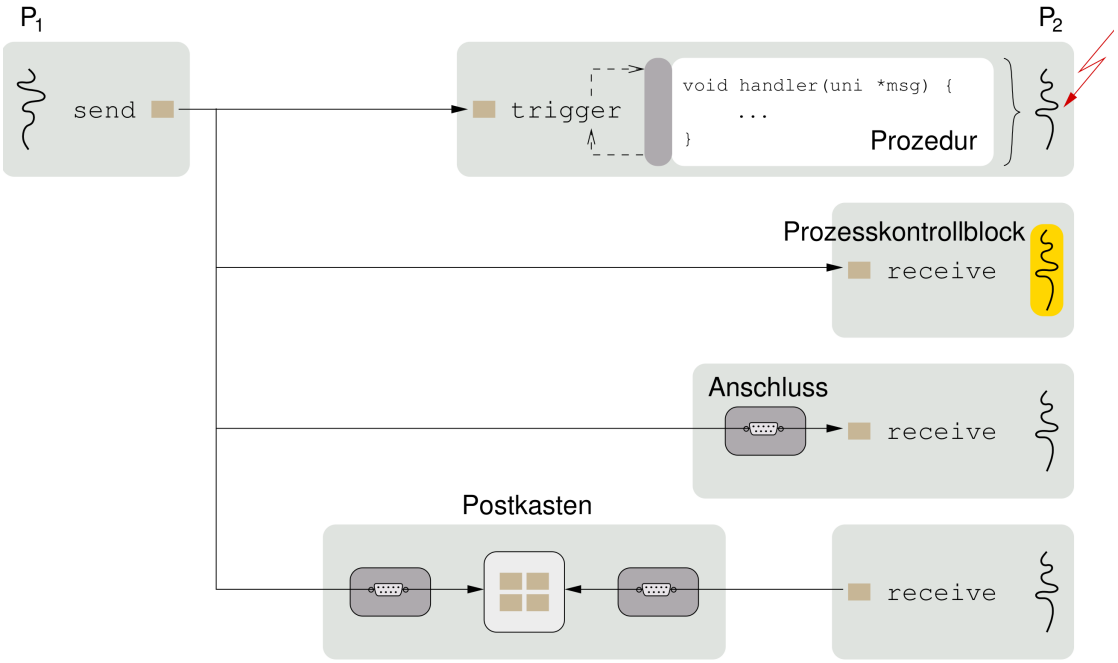


Abbildung 30: Verschiedene Kommunikationsendpunkte im Vergleich

Erkennen der nicht mehr korrekten ID des Empfängers:
Empfänger gestorben und wurde neu erzeugt, erhält dabei eine andere ID
oder Empfänger gestorben, ein anderer Prozess wurde erzeugt, der die selbe ID erhält, wie zuvor der Empfänger.

gerichtete Kommunikation:

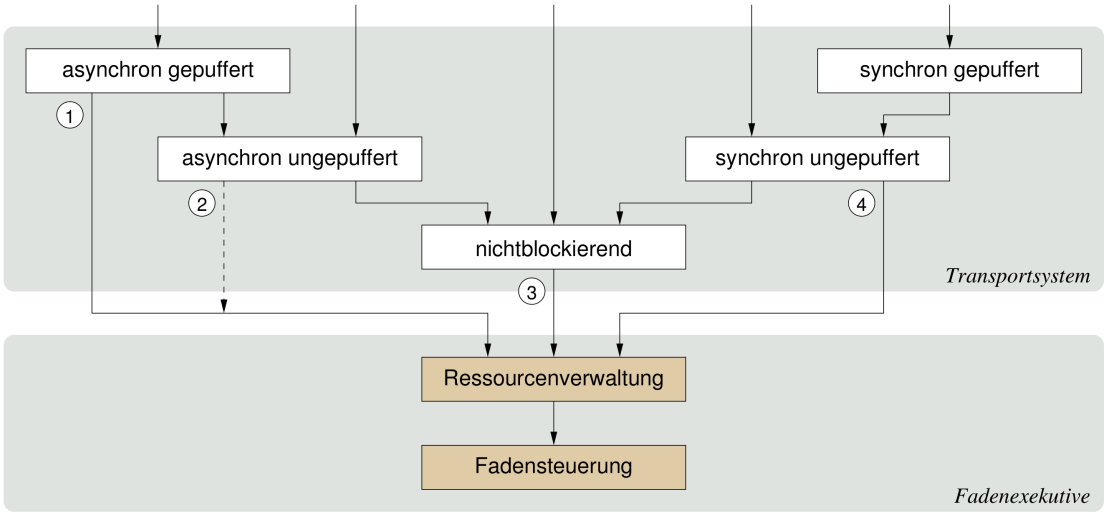
Kennt einen Start- und Endpunkt, zwischen denen eine zeitlich begrenzte feste Verbindung besteht. Vor Senden muss diese aufgebaut, und anschließend wieder abgebaut werden. Dadurch entsteht eine explizite Verbindungsbeziehung, die ein Ausdruck von Datenabhängigkeiten sind, aber dennoch eine Verklemmungsvorbeugung unterstützen.

Aufgrund der Eindeutigkeit der Start-/Endpunkte innerhalb des jeweiligen Prozesses, verfügen die Prozesse über eine *Migrationstransparenz*, sind also unbemerkt verschiebbar. Diese Örtlichkeit muss über den Namensdienst Aufgelöst werden: $name \mapsto (site, port)$

- Startpunkt
  - Anschluss beim Sendeprozess
  - lokale Adresse im Sendeprozess
- Endpunkt
  - Anschluss beim Empfangsprozess
  - lokale Adresse im Empfangsprozess

9 Kommunikationsabstraktionen

In diesem Foliensatz sind viele kleine Beispiele bezüglich Nachrichtenversand und -empfang, die ich jedoch aufgrund der vielen dafür notwendigen Strukturen und der geringen Menge an Code, der unerwartetes enthält, nicht in dieser Zusammenfassung aufgenommen habe.



(a) Ressourcenverwaltung zur Fadenexekutive (Schnittstelle)

1. Bereitstellung eines Puffers zur Nachrichtenaufnahme abwarten
2. Bereitstellung eines Pufferdeskriptors zur Nachrichtenerfassung abwarten
3. Signalisierung der Verfügbarkeit eines Puffers/Pufferdesriptors
4. Signalisiere Empfang einer Nachricht

Asynchrone Kommunikation:

Beide Verfahren wirken blockierend auf den Sendeprozess, sollten die Nachrichtenpuffer oder -deskriptoren ausgegangen sein und dieser Betriebsmittelmangel keine scheiternde Ausnahmesituation hervorrufen.

Synchron kann nur eine Nachricht auf einmal erzeugen, im asynchronen Fall sind beliebig viele gleichzeitig möglich.

- gepuffert
  - eigentlich langsam, allerdings können Wechselpuffer verwendet werden, durch die das ganze relativ schnell werden kann, da die Kopiervorgänge eingespart werden können.
  - Nach Übergabe an das Transportsystem können die Puffer sofort weiter verwendet werden.

- ungepuffert
  - Wiederverwendung des belegten Speicherbereichs ist nach Übergabe an das Transportsystems beschränkt möglich.
  - Die Möglichkeit der Wiederverwendung wird durch den Sendeprozess bei Abschluss signalisiert.

Synchrone Kommunikation:

Beide Verfahren wirken auf Sendeprozess *immer* blockierend. Die Operationen können nicht aufgrund von Betriebsmittelmangel scheitern, da der Sendeprozess niemals mehrere Kommunikationsvorgänge zu einem Zeitpunkt auslösen kann. Scheitern ist jedoch möglich, sofern der Empfangsprozess ungültig ist.

Synchron kann nur eine Nachricht auf einmal erzeugen, im asynchronen Fall sind beliebig viele gleichzeitig möglich.

- gepuffert
  - Auslagerung des Programms unproblematisch. - Bei Kopiervorgang ins Transportsystem - z.B. wenn der Adressraum des Transportsystems kleiner ist und der Rest ausgelagert werden kann
  - sofortige Wiederverwendbarkeit des Puffers möglich, sobald Daten auf dem Transportsystem
- ungepuffert
  - Auslagerung des Programms nur bedingt möglich.
  - Transfer kann Ende-zu-Ende erfolgen, sobald die Daten an das Transportsystem übergeben wurden

Nichtblockierende Kommunikation:

Lieferung der Betriebsmittel stets von oben. Pufferdeskriptoren verweisen auf Puffer, auf die die Nachrichten abgebildet werden

- Puffer
  - Von der Anwendungsebene oder der höheren Systemebene, die eine *gepufferte* Kommunikation implementiert
- Pufferdeskriptor
  - Von der höheren Systemebene, die eine *ungepufferte* Kommunikation implementiert

```
1 struct pack{
2    union{
3       char data[PACK_SIZE]; //bounded dataset
```

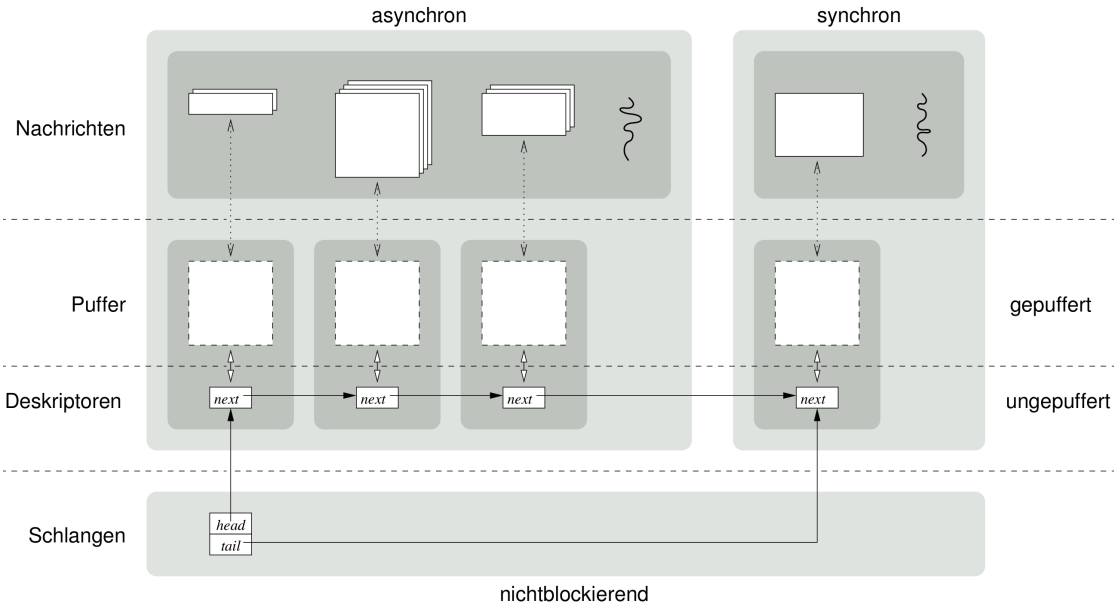


Abbildung 32: Vergleich verschiedener Kommunikationsarten
Prozesse blockieren, unabhängig von der Kommunikations-Synchronität, bei Betriebsmittelmangel, in diesem Fall Nachrichtenpuffer und -deskriptoren.

```
4         section part[PART_COUNT]; //descriptor set
5     }
6 };//cache alignment beachten!
```

Listing 7: Arrays in Union - Packet
Die verschiedenen Arrays/Pointer müssen aufgrund des Unions nicht gecastet werden.

Fadensemaphore:

Eine Semaphore, die fest mit einem Faden verbunden ist. Damit kann für asynchronen Datentransfer signalisiert werden, wann Betriebsmittel verfügbar sind. Entsprechend werden auch prozessübergreifende Prozeduraufrufe unterstützt, indem ein *Versprechen, promise* zur Ergebnislieferung gegeben wird, die *letzendlich (eventual)* beim aufrufenden Prozess eintrifft und Erwartungen an eigene Berechnungen in der *Zukunft (future)* weckt.

Mikrokerne in parallelen Rechnern:

Die reinen Mikrokerne eignen sich nicht vollständig für parallele Systeme und es wird seit jeher auf IPC zurückgegriffen. Beispielsweise in Systemen mit distributed Memory (verteilten Speicher), da die setup time (minimale Rüstzeit) für die Kommunikation wichtig ist. globale IPC sind hierbei unverzichtbar, wohingegen lokale IPC selten auftreten.

Einplanung von Fäden erfolgt auf Vielkernern zunehmend kernlokal, ohne jedoch die kernlokale IPC zwischen Fäden zu implizieren. Zu beachten sind allerdings die Cache-lines, da diese die Größe und Ausrichtung der Nachrichtenpuffer bestimmen.

10 Mitbenutzung

Segmented Paging:

Paging kann segmentiert werden, also eine Page wird unterteilt.
Segmente bestehen aus mehreren Pages und Bruchteilen von Paging - weder externe, noch interne Segmentierung

- $\Im$ sind die N Bereiche in den M logischen Adressräumen, $N \geq M$
- $\Re$ der Abbildungsbereich des realen Adressraums, auf dem $\Im$ abgebildet wird.
- Es gilt $\Im \subseteq \Re$, $\Im$ kann über verschiedene Ausschnitte dieser Adressbereiche überlappen
- $\Im$ und $\Re$ werden gemäß ihrer Granulatgröße aligned

Positions(un)abhängigkeit:

- absolute Adressen erzeugen Positionsabhängigkeiten - meist performanter
- relative Adressen erzeugen Positionsunabhängigkeit (position independent code) - also der Code ist verschiebbar - mapping der Prozesse in den logischen Adressraum
- bei Mehrfacheinblendung eines Bereichs $\Re$ liegt dieser an verschiedenen Bereichen $\Im$ im selben logischen Adressraum - fig. 33

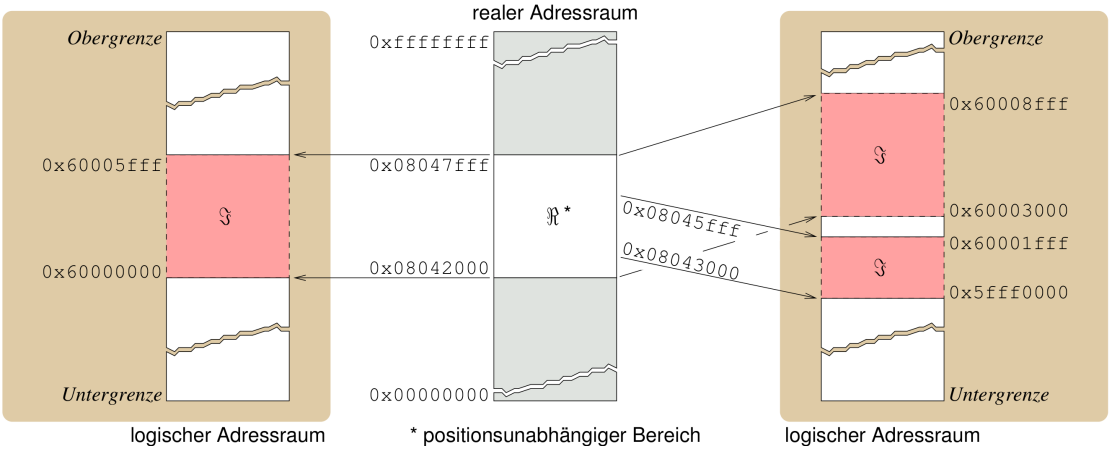


Abbildung 33: Mehrfacheinblendung - Praktisch beispielsweise wenn verschiedene Datenstrukturen nacheinander referenziert werden. Meistens bei parallelem Ablauf. Zum Datenzugriff werden auf $\Re$ verschieden breite Fenster gelegt und die Prozesse entscheiden selbst, wo die Einblendung des Bereichs $\Re$ erfolgt.

Systemeigenschaft der Text- bzw. Datenverbünde (code sharing / data sharing):

- statisch
  - $\mapsto$ Indivualbibliothek
  - Binden der Symbole an Adressen vor Laufzeit, diese Bindung ist zur Laufzeit underveränderlich
- dynamisch
  - Binden der Symbole an Adressen zur Laufzeit, diese Bindung ist i.A. nach der ersten Laufzeitreferenz fest.
  - Optimalerweise Adressunabhängig.

copy on write (COW - kopieren beim Schreiben):

Der Grundgedanke ist es kopierte Daten erst dann tatsächlich zu kopieren, wenn sich Teile entweder des Originals oder aber der Kopie ändern würden. Somit können logisch zwei verschiedene Datensätze physikalisch einmal im Speicher vorgehalten werden. Beide logischen Datensätze werden als read only markiert und bei dem Versuch schreibend darauf zuzugreifen wird das Betriebssystem eingeschaltet, welches den entsprechenden Teil kopiert und dann den betreffenden Teil in einem der beiden Datensätze überschreibt.

Prozesse die Text-/Datenbereiche mitbenutzen dürfen, erhalten implizit Wissen über die Existenz abgebildeter Objekte im eigenen Adressraum, die den Prozessen logisch, nicht jedoch real als *Eigentum* überlassen wurden.

Die Schreibrechte werden quellseitig entzogen und zielseitig nicht erteilt. Dadurch wird eine Momentaufnahme (Snapshot) des Zustands zum Schreibzeitpunkt erzeugt. Nach dem Schreibvorgang besitzen die Prozesse Schreibrechte auf Original und Kopie. Der schreibende Prozess legt eine eigene Version des Objekts an, die an die originale Adresse des eigenen logischen Adressraums gebunden ist. - Beispiel figs. 34 to 38

Verkettung der Deskriptoren, damit bei einem Page Fault schnell alles relevante angefasst werden kann. Weil eine Änderung bei sich auch eine Änderung bei den anderen nach sich ziehen muss.

copy on reference (COR - kopieren beim Referenzieren):

In dem Adressraum des Zugreifers wird eine Objektabbildung erstellt, die zielseitig mit Lese-, Schreib- oder Ausführungsrechten versehen wird. Quellseitig hingegen werden Schreibrechte entzogen, also auf COW umgestellt.

Kopplung mit IPC zur empfangsseitigen Nachrichten-Einblendung:

- reliable-blocking send

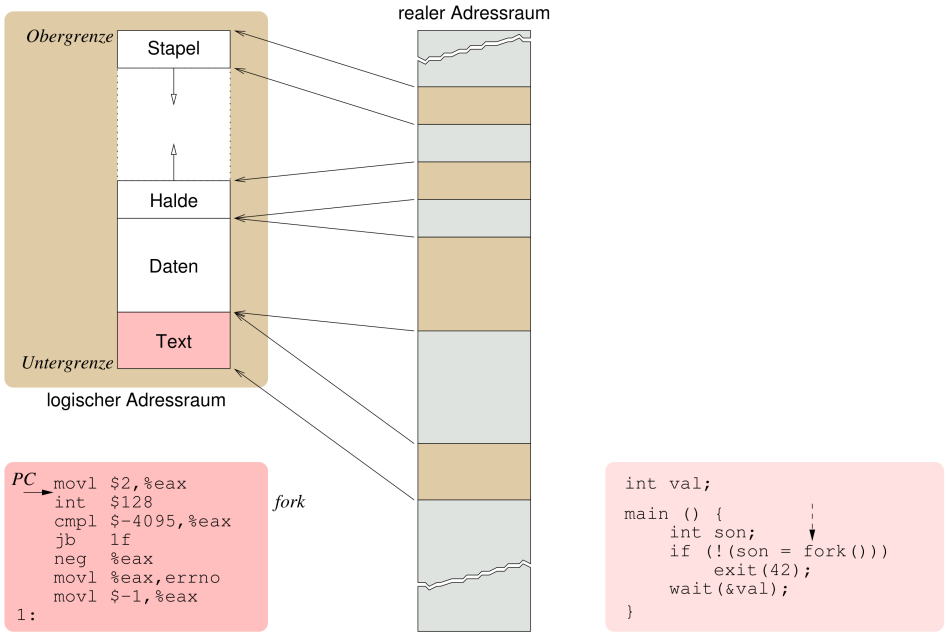


Abbildung 34: COW - Prozessinkarnation 1 - figs. 34 to 38 - etwas detaillierter findet sich das hier ff

  - Annahme im BS
  - Sender gibt Verortung vor
- synchronization send
  - receive
  - Empfänger gibt ggf. Verortung vor
- remote-invocation send
  - receive
  - Empfänger gibt Verortung vor
  - explizit zwischen receive und reply

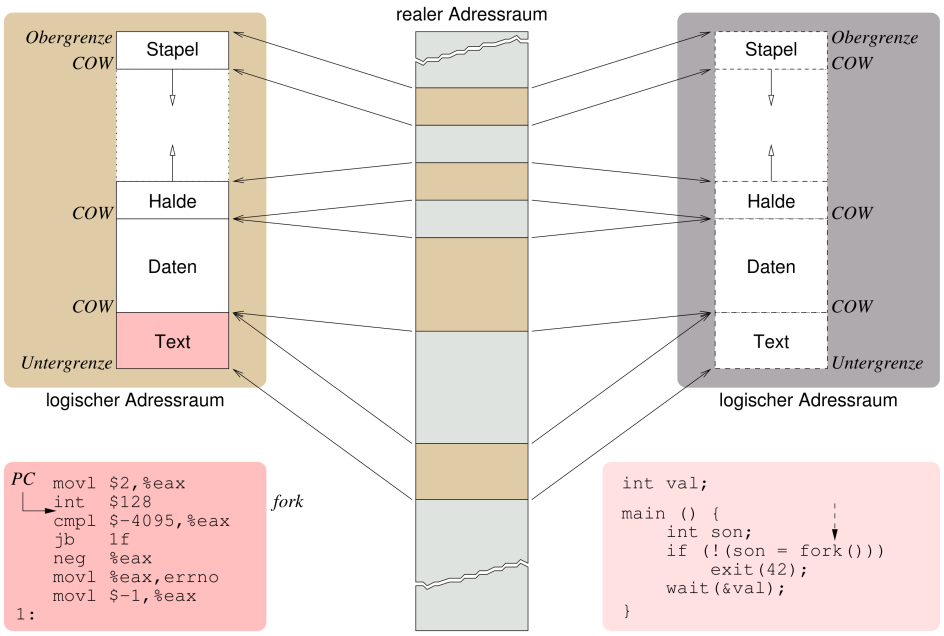


Abbildung 35: COW - Prozessinkarnation 2 - figs. 34 to 38 - etwas detaillierter findet sich das hier ff

Logik hinter einem IPC-Deskriptorversand (send) - figs. 42 to 44:

1. Empfänger identifizieren
2. Sender identifizieren
3. Nachricht zustellen
4. Erwarte Ankunft

- bei direkter Kommunikation des Nachrichtendeskriptors durch den Sender: Entleerung des Puffers durch den Empfänger abwarten
- bei indirekter Kommunikation der Nachricht durch den Empfänger: wird erst bei Bedarf durch COW(Sender) oder COR(Empfänger) übertragen

Logik hinter einem IPC-Deskriptorversand-/anwendung (emit):

Ist dem *send* sehr ähnlich, jedoch ist *emit* speicherabbildend und bedingt synchronisiert. Der Sender reserviert einen Pufferbereich im Adressraum des Empfängers. - Beispiel figs. 39 to 41

1. Empfänger identifizieren
2. Sender identifizieren
3. Verknüpfe COR/COW-Map
4. Belege Puffer
5. Nachricht zustellen

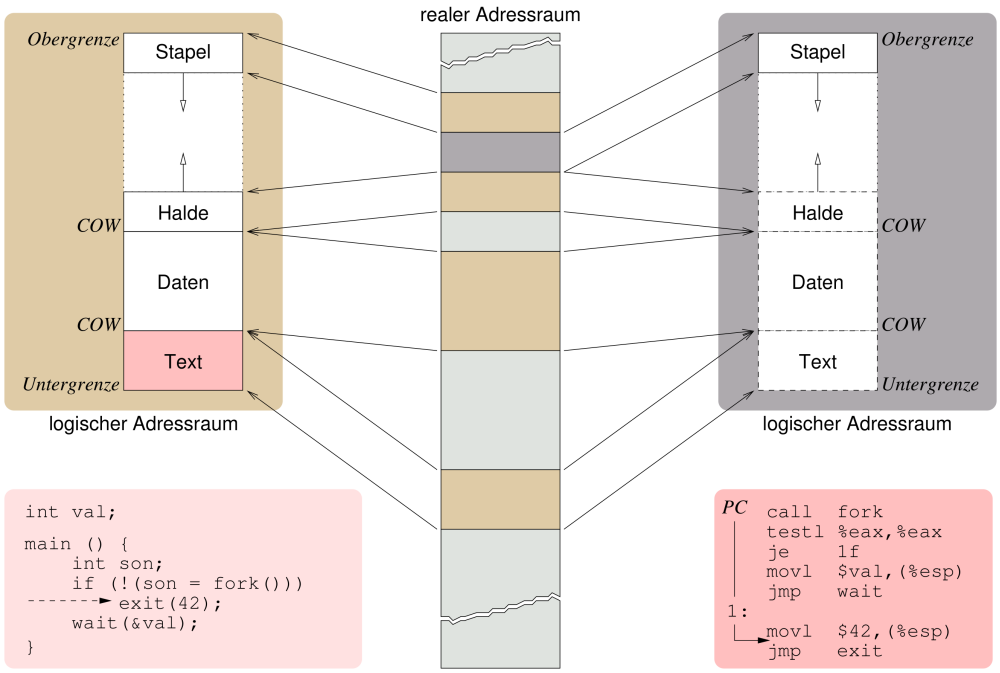


Abbildung 36: COW - Prozessinkarnation 3 - figs. 34 to 38 - etwas detaillierter findet sich das hier ff

Logik hinter einem IPC-Deskriptorempfang-/anwendung (receive): Beispiel figs. 42 to 44

1. Empfänger identifizieren
2. Akzeptiere Nachricht
3. falls Nachrichten mapped: Puffer freigeben
4. ansonsten, wenn nicht Nachrichten mapped: behalte die Nachricht

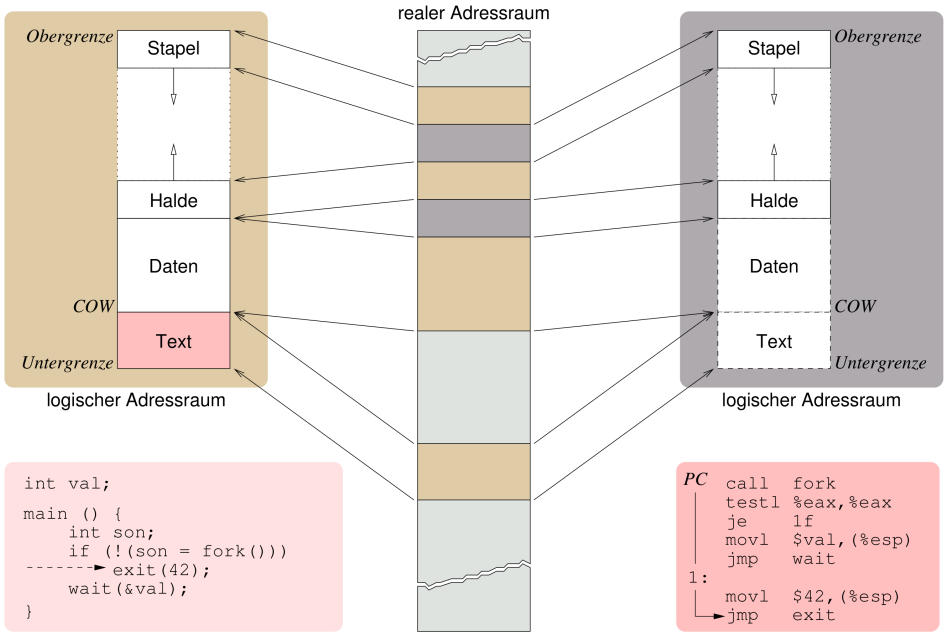


Abbildung 37: COW - Prozessinkarnation 4 - figs. 34 to 38 - etwas detaillierter findet sich das hier ff

Logik IPC-Deskriptoranwendung (serve):

Beispiel figs. 39 to 41, mit etwas anderem Code - genauer hier ff

1. Empfang des Auftrags als Nachrichtendeskriptor (receive)
2. Nachrichtenverarbeitung durch Anwendung des Deskriptors (apply) - analog zu emit oder receive - allerdings explizit im Anwendungsprozess
   a) Aufgerufenen identifizieren
   b) Aufrufer identifizieren
   c) Puffer im Adressraum des Aufrufers reservieren
   d) Zieladressbereich zuweisen (allot)
   e) Verknüpfe COR/COW-Map
   f) Quelladressbereich abändern (alter)
3. Rückantwort und Übermittlung einer Ergebnisnachricht (reply)

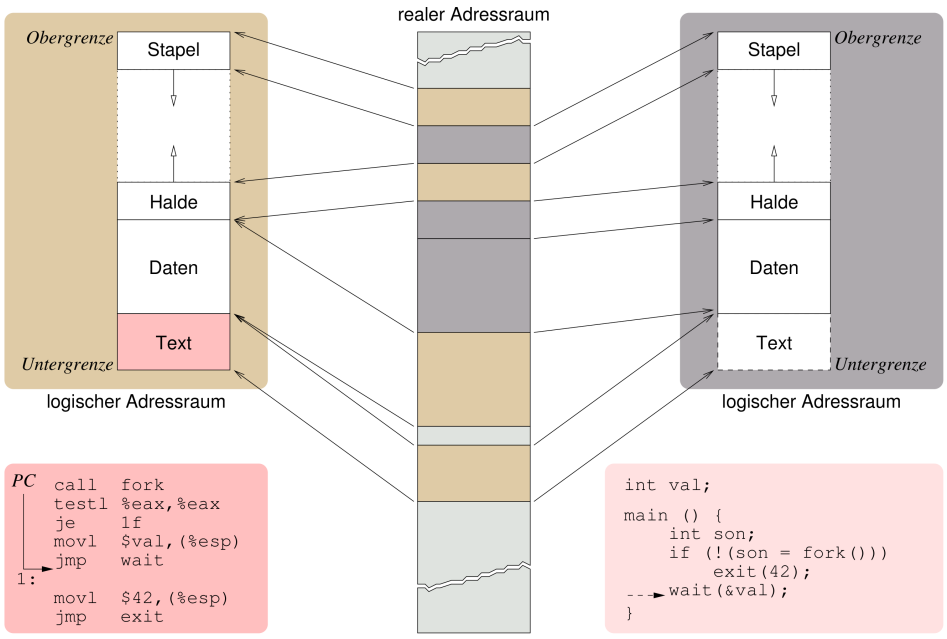


Abbildung 38: COW - Prozessinkarnation 5 - figs. 34 to 38 - etwas detaillierter findet sich das hier ff

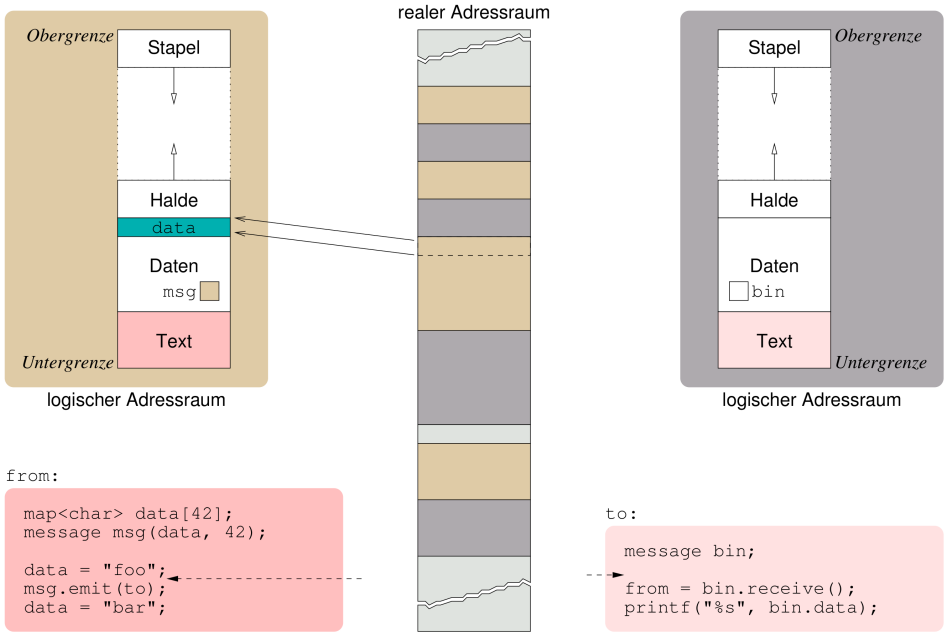


Abbildung 39: emit/receive - Deskriptorversand/-anwendung 1 - figs. 39 to 41 - etwas detaillierter findet sich das hier ff

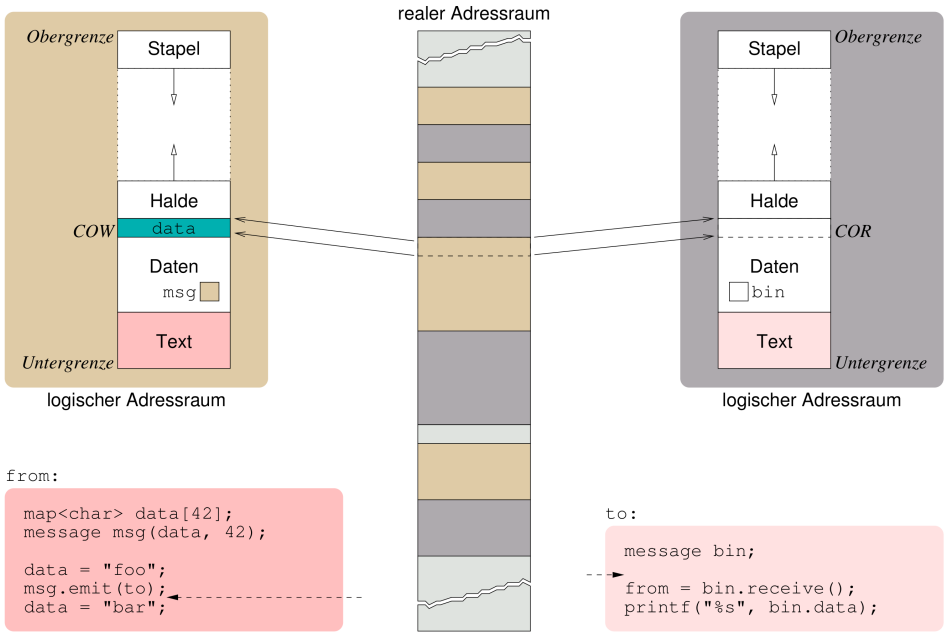


Abbildung 40: emit/receive - Deskriptorversand/-anwendung 2 - figs. 39 to 41 - etwas detaillierter findet sich das hier ff

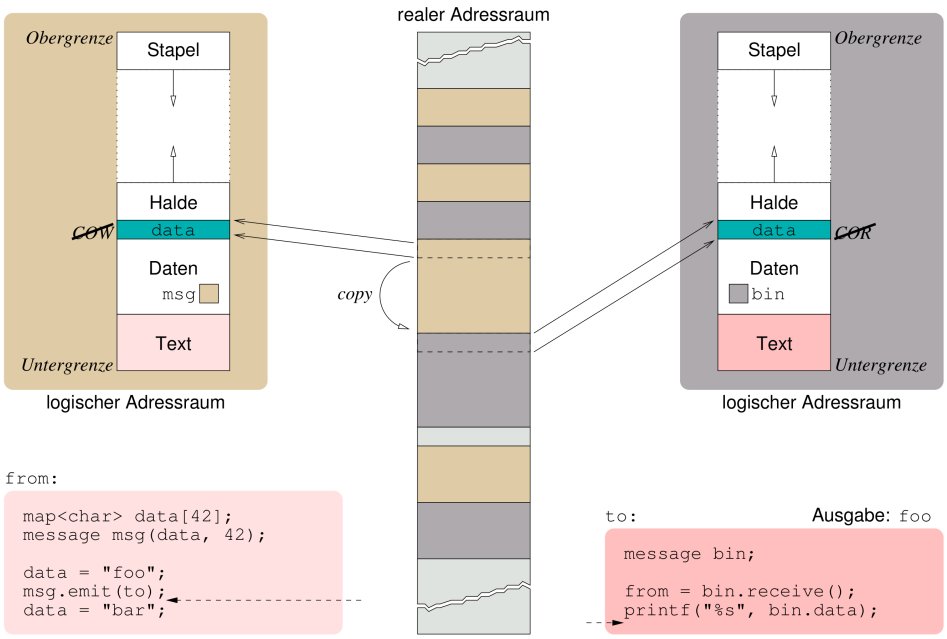


Abbildung 41: emit/receive - Deskriptorversand/-anwendung 3 - figs. 39 to 41 - etwas detaillierter findet sich das hier ff - Kopie wird beim Lesenden Zugriff angelegt

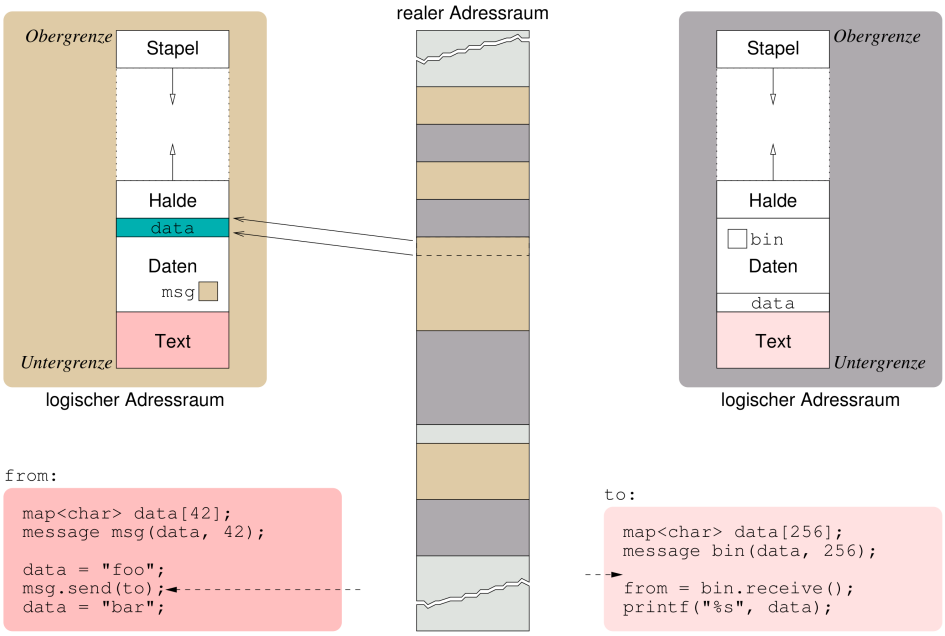


Abbildung 42: send/receive - Deskriptorempfang/-anwendung 1 - figs. 42 to 44 - etwas detaillierter findet sich das hier ff

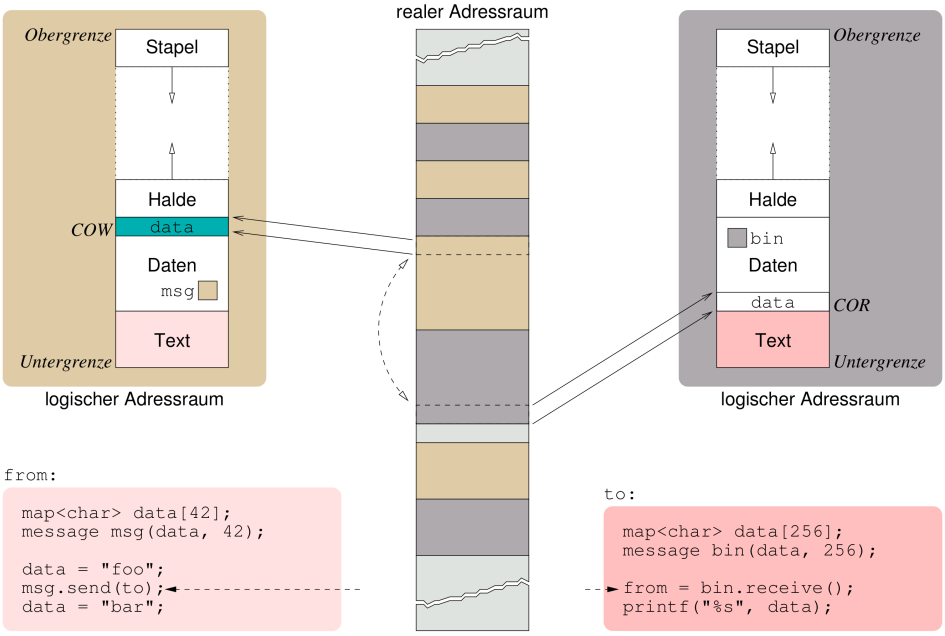


Abbildung 43: send/receive - Deskriptorempfang/-anwendung 2 - figs. 42 to 44 - etwas detaillierter findet sich das hier ff

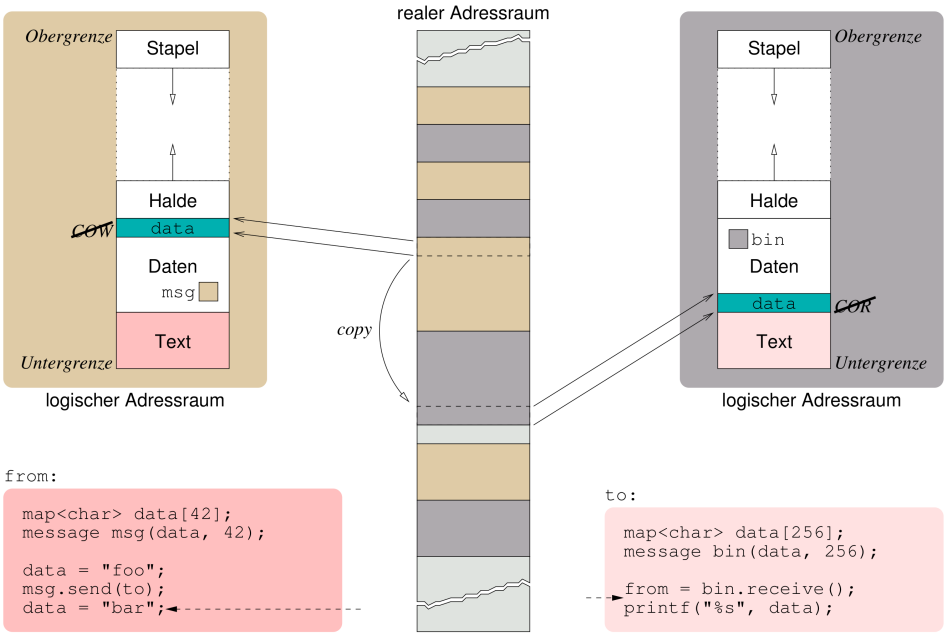


Abbildung 44: send/receive - Deskriptorempfang/-anwendung 3 - figs. 42 to 44 - etwas detaillierter findet sich das hier ff

11 Bindelader

Programmbibliothek:

Ist ein Bündel von Unterprogrammen oder Objekten, die von Programmen durch symbolische Adressierung angefordert werden.

statische Bindung:

- Binden der Symbole an Adressen erfolgt vor Lade- oder Laufzeit
- Die Verknüpfung zu den Objekten erfolgt allerdings erst zur Laufzeit (in der Zwischenzeit sollte die Bibliothek nicht manipuliert werden - Vorbeugen: Versionierung oder indirekte Adressierung mit Sprungtabelle oder Gemeinschaftsblock)
- vergrößert Programme - als Folge großer Speicherplatzbedarf im Vorder- und Hintergrund
- Bilbiotheksänderungen wirken sich nicht auf gebundene Programme aus
- Idividualbibliothek, wird repliziert gespeichert
- Symbolauflösung erfolgt durch den Linker, indem der Linker zur Bindezeit in den Bibliotheken nach Objektmodulen sucht, die undefinierte externe Symbole auflösen (exportieren). Für diese gefundenen Module vermerkt er in welcher Bibliothek diese enthalten sind in einer Liste der Bibliotheken innerhalb des Lademoduls (executable)
- profitiert nicht von positionsunabhängigen Code, im Gegenteil es sorgt für Overhead.

dynamische Bindung:

- erfolgt zur Lade- oder Laufzeit
- Ladezeit: im Moment der Einblendung in den logischen Adressraum, zur Laufzeit können absolute Adressen gespeichert werden
- Laufzeit: im Moment des Befehlsabrufs aus dem logischen Adressraum, es dürfen nur relative Adressen gespeichert werden
- kleinere Programme liegen im Hintergrund (kleine Lademodule)
- kleinerer Speicherplatzbedarf im Vordergrund (dynamische Bindung, nur relevante Teile)
- Bibliotheksänderungen wirken sich auf Programme aus
- Gemeinschaftsbibliothek, ist im Arbeitsspeicher ggf. repliziert

- Symbolauflösung durch den Loader, indem der Loader zur Ladezeit für das Programm einen logischen Adressraum aufsetzt und anschließend einen startup code (Anlaufprozedur) durchläuft, die Bibliotheken findet und in den Programmadressraum einblendet.

Einlagerung von Bibliotheken in den realen Adressraum:

- im Voraus (statische Gemeinschaftsbibliothek)
  - vor Programmstart
  - Laden aller Objekte gebundener Symbole, die noch nicht geladen wurden
  - jede Gemeinschaftsbibliothek definiert einen festen Adressbereich, diese müssen mit anderen Bibliotheken überlappungsfrei sein. Das kann theoretisch statisch überprüft werden, allerdings ist Adressraumzuweisung zu Bibliotheken schwarze Magie
- auf Anforderung (dynamische Gemeinschaftsbibliothek)
  - nach Programmstart - Ein Großteil des Link-Vorgangs wird bis zur Startzeit aufgeschoben (oder noch weiter)
  - speicherabgebildete Datei (memory-mapped file)
  - Ist die von der Bibliothek gewünschte Adresse bereits belegt, wird versucht eine andere passende zu finden. Ist dies nicht möglich, so scheitert das Einblenden.
  - Möglichkeiten
    - explizit
      - durch programmierbares Nachladen (dlopen)
      - in jeder Hinsicht für den Prozess intransparent
      - ausgelöst durch Prozedur- oder Systemaufruf
    - implizit
      - durch partielle Interpretation
      - in funktionaler Hinsicht transparent für den Prozess
      - ausnahmebedingt ausgelöst durch synchrone Programmunterbrechung

Positionsaunabhängigkeit:

Legt keine Programmbereiche fest, wie es bei positionsabhängigkeit der Fall wäre. Allerdings ist eine Relokation zur Laufzeit, gleichwohl möglich, zu aufwändig. Denn es müssen alle zu ändernden Programmadressen bekannt sein: Für Programmtext stehen diese in der Symboltabelle des Linkers und für Programmdaten sind die dynamischen Datenstrukturen zu verfolgen. Schwierig hingegen ist dies bei dynamischen Datenstrukturen des Laufzeitsystems (Halde, Stack, ...), deren Adressen nicht wirklich offen liegen.

Die Programme müssen frei verschiebbar sein, also ausschließliche Verwendung relative Adressierung im Programmtext.

Auswirkungen von Positionsunabhängigen Programmen auf Gemeinschaftsbibliotheken:
Positionsunabhängige wirkt sich negativ auf die Laufzeit von Gemeinschaftsbibliotheken aus

- Ladezeit, mögliche Optimierung ist Zwischenspeicherung
  - Verschieben der Bibliothek (relocation)
  - Auflösung von Programmsymbolen (resolution)
- Laufzeit, mögliche Optimierung ist nur das Aufgeben der Positionsunabhängigkeit
  - Entschleunigung der Einsprungtabelle
  - Mehraufwand durch Funktionensprolog (Berechnung abhängig vom Programmcounter, der zuvor gesichert werden muss)
  - indirekte Datenreferenzen
  - reservierte Adressregister werden verlangsamt

11.1 Multics

Findet wohl derzeit keine Verwendung mehr.

Idee:

- Dateien und Arbeitsspeicher werden zu einem einstufigen Speicher verschmolzen
- bedarfsorientierte Programmausführung durch dynamisches Binden
- benutzerorientiertes hierarchies Dateisystem
- ringorientierter Schutz
- mitlaufende Hardware-Rekonfigurierung
- in Multics liegt im Grunde alles in einem eigenen Segment, auch die eigenen Programme. Beispiel Shell soll ein excec auf ein anderes Programm ausführen, dann wird schlicht in das entsprechende Segment gesprungen.
- Alles kann dynamisch gebunden werden, auch die eigenen Programmstrukturen
- Sämtliche Informationen sind direkt über Segmente referenzierbar, in anderen Systemen ist dies ausschließlich über Dateien möglich
- Referenzlokalität (locality of reference) stehen im Vordergrund, denn segmentlokale Referenzen bedingen keiner indirekt adressierten Wortpaare
- alle Prozeduren liegen in einer Gemeinschaftsbibliothek

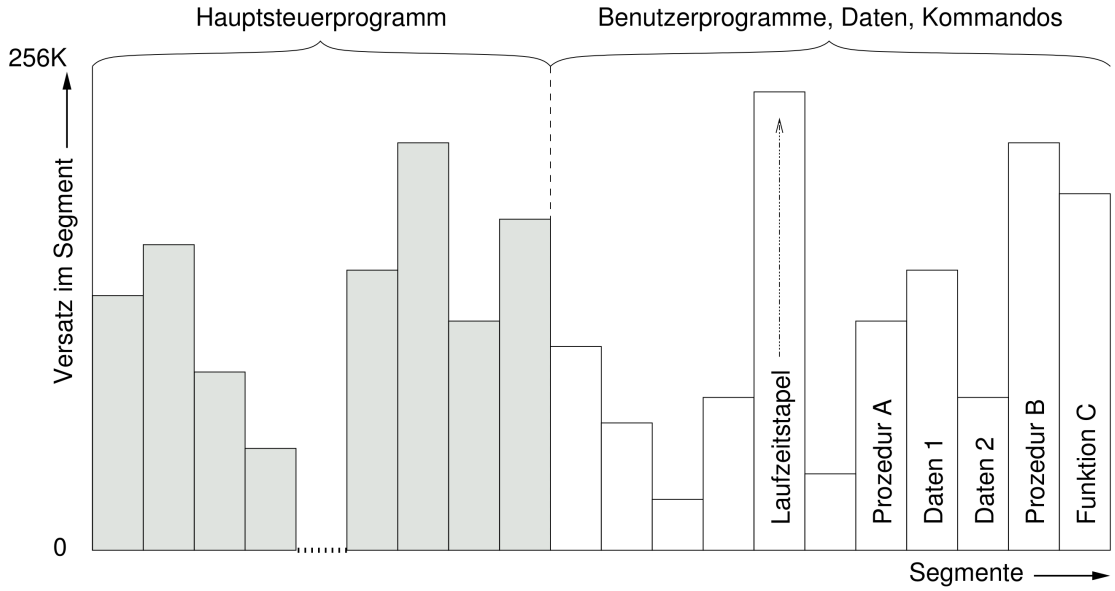


Abbildung 45: Zweidimensionaler Prozessadressraum in Multics

Adressbasisregisterpaare:

Die Adressbasisregisterpaare steuern ein Kontrollfeld im Adressbasisregister (ABR), wobei 0 segmentlokale Wortadresse und 1 globaler Segmentname (externe Basis) signalisiert.

- Argumentliste (argument pointer/base), AP & AB, 0-1
- allgemeine Basis (base pointer/base), BP & BB, 2-3
- Bindungssegment (lingage pointer/base), LP & LB, 4-5
- Stapelsegment (stack pointer/base), SP & SB, 6-7

Adressbildung:

Adressen bestehen aus einer Segmentnummer und einer Wortnummer innerhalb des Segments, diese sind ortsunabhängig. Daraus wird eine *generalisierte Adresse* gebildet: Basisadresse der Segmentdeskriptortabelle (descriptor base register, DBR), Segmentnummer der aktuellen Prozedur (procedure base register, PBR) und vier Adressbasisregisterpaare (AP, BP, LP, SP)

Die einzelnen Prozessadressräume werden jeweils durch ein Deskriptorsegment repräsentiert, wobei DBR die Anfangsadresse dieses Segments im Hauptspeicher enthält.

Im Grunde ist dies Segmentierung wobei Segmentnummer $\mapsto$ Segmentname und Wortnummer $\mapsto$ Wortadresse

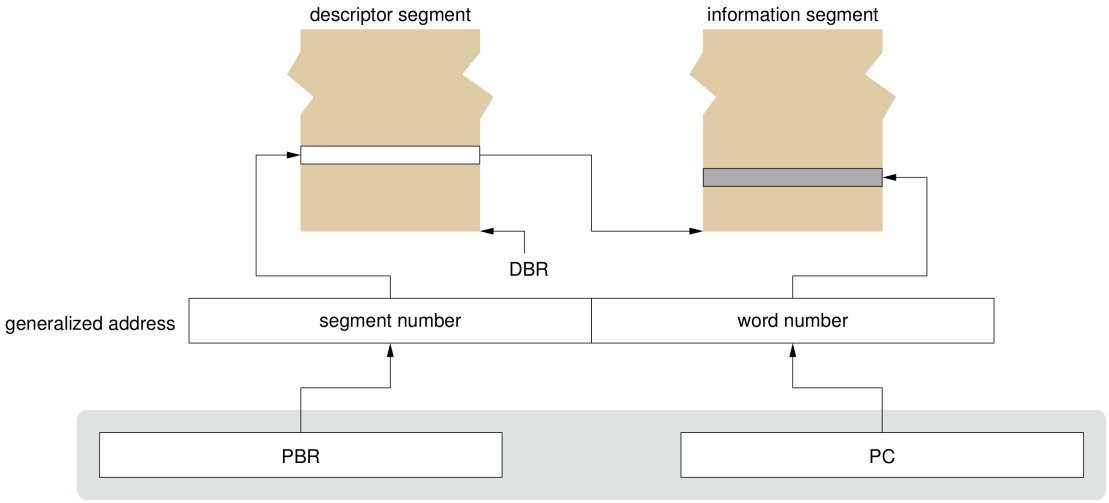


Abbildung 46: Adressbildung in Multics - Adresse für einen Befehlsabruf wird konkateniert aus der Segmentnummer (DBR) und der Wortnummer (PC)

ITS Pointer:

Der Zeigerzustand wird vom Etikett im Leitwort definiert

- snapped $\mapsto$ Snapshot (ITS)
  - Zeiger ist an eine generalisierbare Adresse gebunden
  - das ist der Normalfall, weshalb der Prozessor den Zugriff direkt ausführt
  - das Etikett im Folgewort spezifiziert den weiteren Zugriff
- unsnapped (FT2)
  - ungebundener Zeiger
  - Ausnahmefall, der Prozessor löst einen Zugriffsfehler aus
  - Behandlung des Bindefehlers (link trap) erfolgt durch das Betriebssystem
  - Der Prozessor wiederholt dern Zugriff nach Wiederaufnahme

Etablieren einer Verknüpfung (ITS Pointer) im Bindungssegment:

Hierbei wird aus einer *offenen Verknüpfung (unsnapped link)* eine *eingerastete Verknüpfung (snapped link)*. Dies wirkt sich lediglich auf das Bindungssegment des Prozesses aus, nicht auf das Prozedursegment. Letztere sind invariant gegenüber äußeren Einflüssen. Aufgrund des FT2-Etiketts im Leitwort wird ein Bindefehler (link trap) erzeugt und vom Bindelader (linking loader) behandelt

1. Dieser lokalisiert das symbolisch adressierte Segment
2. diese Adresse soll in eine generalisierte Adresse umgewandelt werden
3. das angeforderte Objekt wird anhand dieser erzeugten Adresse in den virtuellen Prozessadressraum eingeblendet

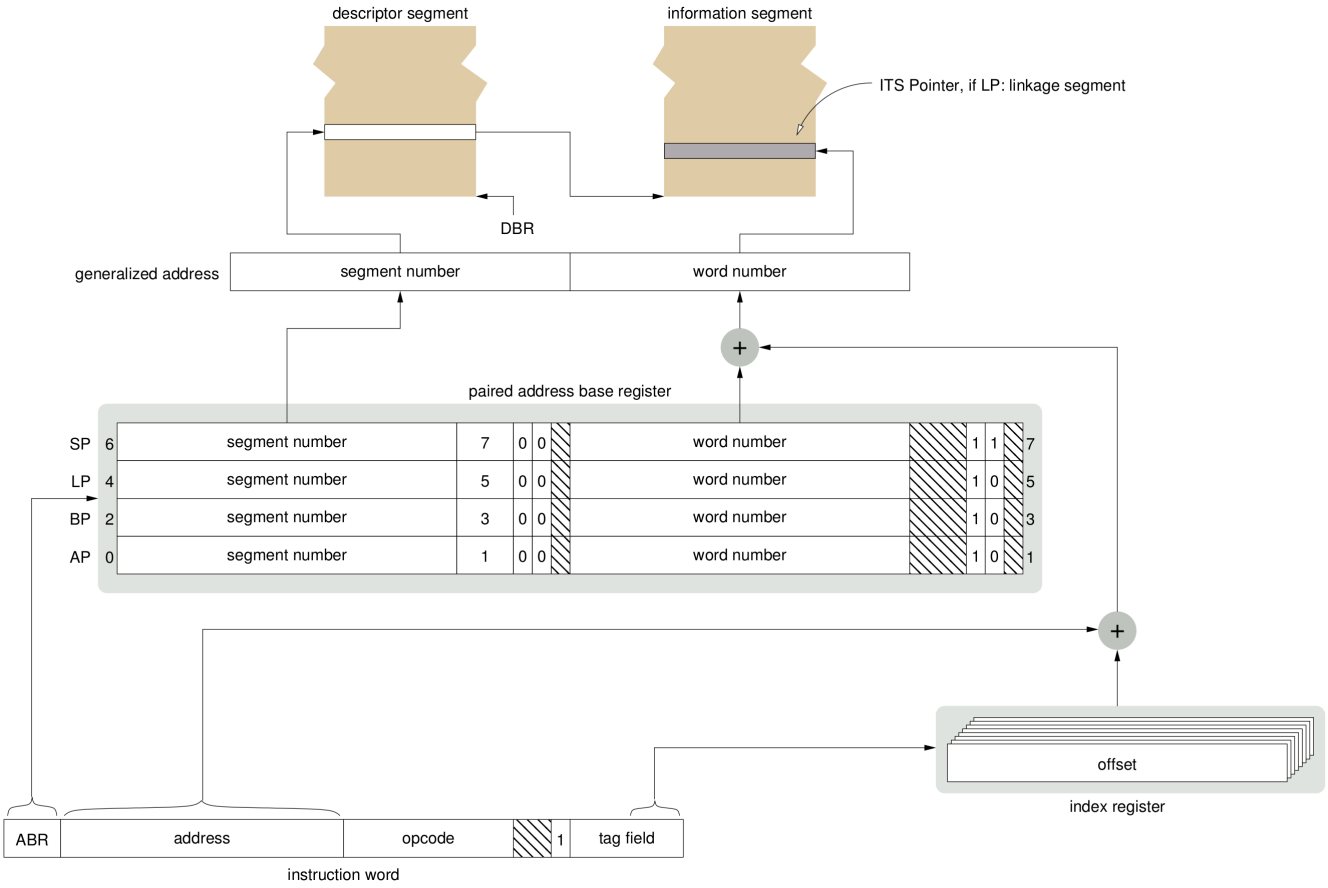


Abbildung 47: Adressbildung in Multics 2

12 Abbildungsverzeichnis

Abbildungsverzeichnis

13 Tabellenverzeichnis

Tabellenverzeichnis

14 Listingverzeichnis

List of Listings