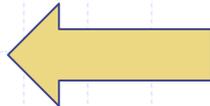


Betriebssysteme Vorlesung 5

Verwaltung des Hauptspeichers
Memory Management I
Anforderungen, Paging, Segmentation

Literatur: Stallings Kapitel 7, Appendix 7 A

Stand der Vorlesung

- ◆ Einführung: Ein Betriebssystem - Was ist das ?
- ◆ Prozeßmanagement
 - Prozesse
 - Threads
 - Wechselseitiger Ausschluß, Deadlocks & Starvation
- ◆ Speicher Management
 - einfache Speicherverwaltung  HEUTE !
 - virtueller Speicher
- ◆ Prozessor Scheduling
- ◆ E/A Management, Festplattenscheduling
- ◆ Datei Management
- ◆ Netzwerke
- ◆ Sicherheit

Übersicht

◆ Grundlagentechniken der Speicherverwaltung

- Feste Partitionierung (fixed partitioning)
- Dynamische Partitionierung (dynamic partitioning)
- einfaches Seitenverfahren (simple paging)
- einfaches Segmentierungsverfahren
(simple segmentation)

◆ Vorgang beim Binden und Laden eines Programms

Nutzen

- ◆ Mit den einfachen Techniken der Speicherverwaltung lassen sich die Verfahren für virtuellen Speicher einfacher verstehen.
- ◆ Sie lernen mit dem Buddy System eine Technik kennen, die im Betriebssystemkernel eingesetzt wird.
- ◆ Kenntnisse über die Vorgänge beim Compilieren, Binden und Laden eines Programms helfen Ursachen von Softwareproblemen und -fehlern zu diagnostizieren und zu beheben.

Speicherverwaltung bei Multiprogramming

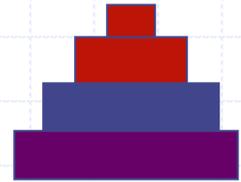
◆ Anforderungen an Speicherverwaltung bei n Prozessen:

- jeder Prozeß muß ausreichend Speicher erhalten
- Versorgung der CPU (schnellste Komponente im System!) mit Instruktionen und Daten möglichst verzögerungsfrei
- die Größe aller Prozeßimages darf den Hauptspeicherplatz überschreiten

◆ dafür verfügbare Ressourcen: Speicherhierarchie

Level 1 & Level 2 Caches, Hauptspeicher, Plattenspeicher, ...

zunehmendes Fassungsvermögen, zunehmende Zugriffszeiten



◆ Lösungen

- Paging: Aufteilen des Hauptspeichers in gleichgroße Partitionen (Seitenrahmen),
ein Prozeß belegt eine variable Anzahl von Seitenrahmen
- Segmentierung: Aufteilen des Hauptspeichers in variabel große Partitionen je nach Bedarf eines Prozesses

Grundidee der Speicherverwaltung, weitere Anforderungen

- ◆ Der **Kernel** erhält einen **festen Anteil** des Hauptspeichers, der Rest wird für Anwendungsprozesse genutzt.
- ◆ Falls Hauptspeicherplatz nicht für alle Prozeßimages ausreicht, werden **Prozeßimages auf die Festplatte ausgelagert**
 - **Swapping**: Auslagern eines ganzen Images
 - **Paging, Segmentierung**: Auslagern von Teilbereichen eines Images
- ◆ aus Multiprogramming resultieren **Anforderungen an OS**
 - **Relocation** (= Ortsveränderung)
 - Position des Prozeßimages im Hauptspeicher ist variabel
 - => Code kann keine physikalischen Adressen für Verweise auf Code und Daten enthalten
 - => Programmierer und Compiler erzeugen nur logische Adressen

weitere Anforderungen

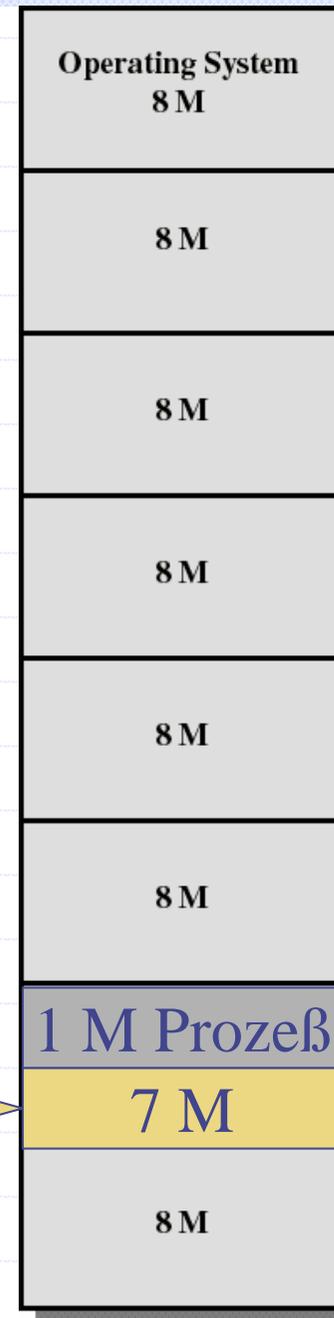
- **Protection** (= Schutz vor externen Zugriffen)
Prozesse sollen nicht unkontrolliert auf Speicherbereiche anderer Prozesse zugreifen dürfen,
wg logischer Adressen nur zur Laufzeit überprüfbar
- **Sharing** (= Teilen von Speicherbereichen zwischen Prozessen)
erlaubt geteilten Speicher (konträr zu Protection)
z.B. für gemeinsame Daten bei kommunizierenden Prozessen,
z.B. für gleichen Programmcode (nur Lesezugriffe)
- **Logische Organisation**
modulare Struktur von Programmen für Protection & Sharing nutzen,
Code Module sind nur ausführbar, (private, public, protected),
Datenmodule können lesbar/schreibbar sein
- **Physikalische Organisation**
Transfer von Speicherinhalten i.w. zwischen Hauptspeicher-
Plattenspeicher

Zunächst einfache Speicherverwaltung ohne virtuellen Speicher

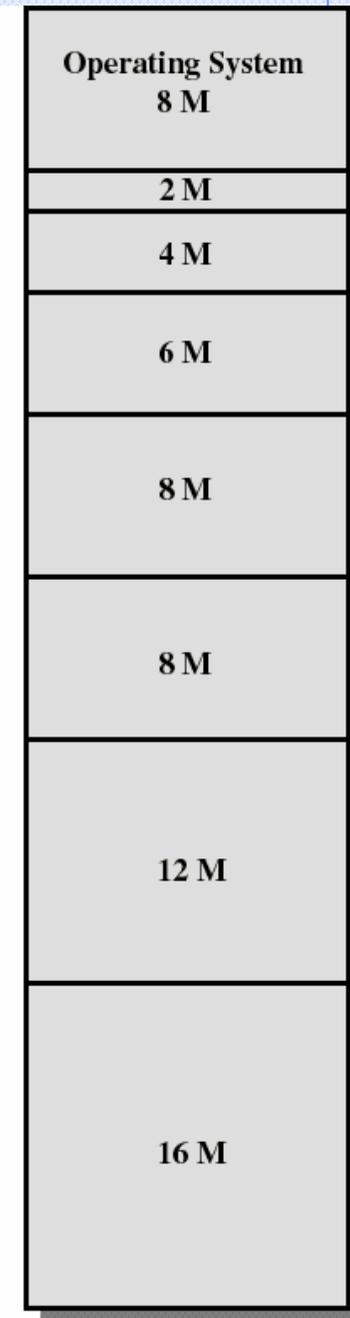
- ◆ Annahme: ausführbare Prozesse werden komplett in den Hauptspeicher geladen
- ◆ wir betrachten folgende Grundlagentechniken
 - Feste Partitionierung (fixed partitioning)
 - Dynamische Partitionierung (dynamic partitioning)
 - einfaches Seitenverfahren (simple paging)
 - einfaches Segmentierungsverfahren (simple segmentation)
- ◆ dadurch ist das Konzept des virtuellen Speichers dann einfacher nachvollziehbar

Feste Partitionierung

- ◆ Hauptspeicher wird in Partitionen fest und disjunkt zerlegt, Partitionsgrößen sind gleich oder ungleich.
- ◆ Prozeß wird in ausreichend große Partition geladen.
 - Nichts frei ? Swapping!
 - Prozeß zu groß ? Overlay-Technik
- ◆ Speichernutzung wg Restplatz in Partitionen schlecht (**interne Fragmentierung**)
- ◆ Austauschverfahren für Prozesse einfach



Equal-size partitions



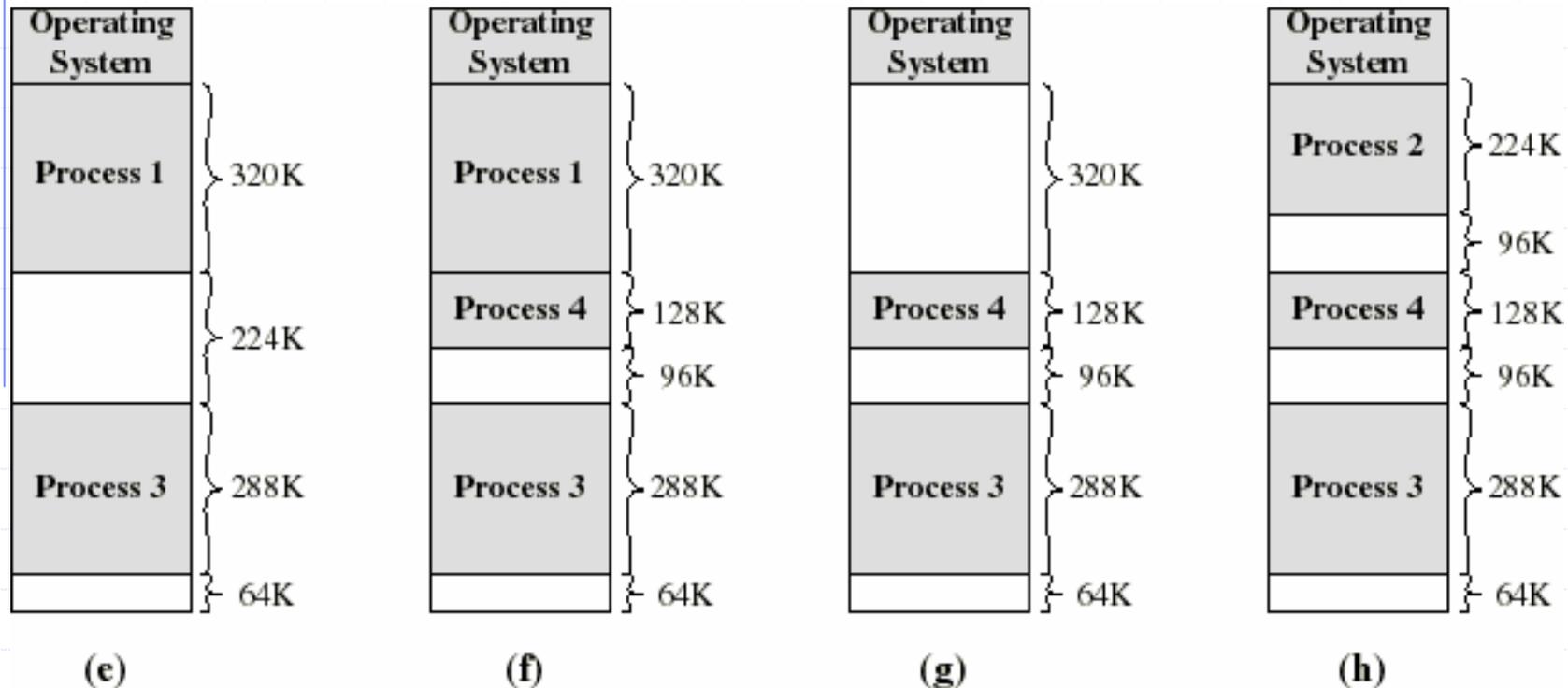
Unequal-size partitions

Dynamische Partitionierung

- ◆ Partitionsanzahl und -größe sind variabel.
- ◆ Jeder Prozeß erhält genau soviel Speicher wie er benötigt.
- ◆ Nachteil: Ungenutzte Bereiche/Löcher zwischen belegten Speicherbereichen entstehen (**externe Fragmentierung**)
- ◆ Gegenmittel: Kompaktifizierung verschiebt belegte Speicherbereiche, damit am Ende freier Bereich entsteht

- ◆ Beispiel:
 - Speicher wird nacheinander mit 3 Prozessen P1 (320 KB), P2 (224 KB) und P3 (288 KB) belegt. Es bleiben 64 KB frei.
 - P2 blockiert und wird auf die Festplatte ausgelagert (Swapping) um Platz für neuen Prozeß P4 (128 KB) zu schaffen.

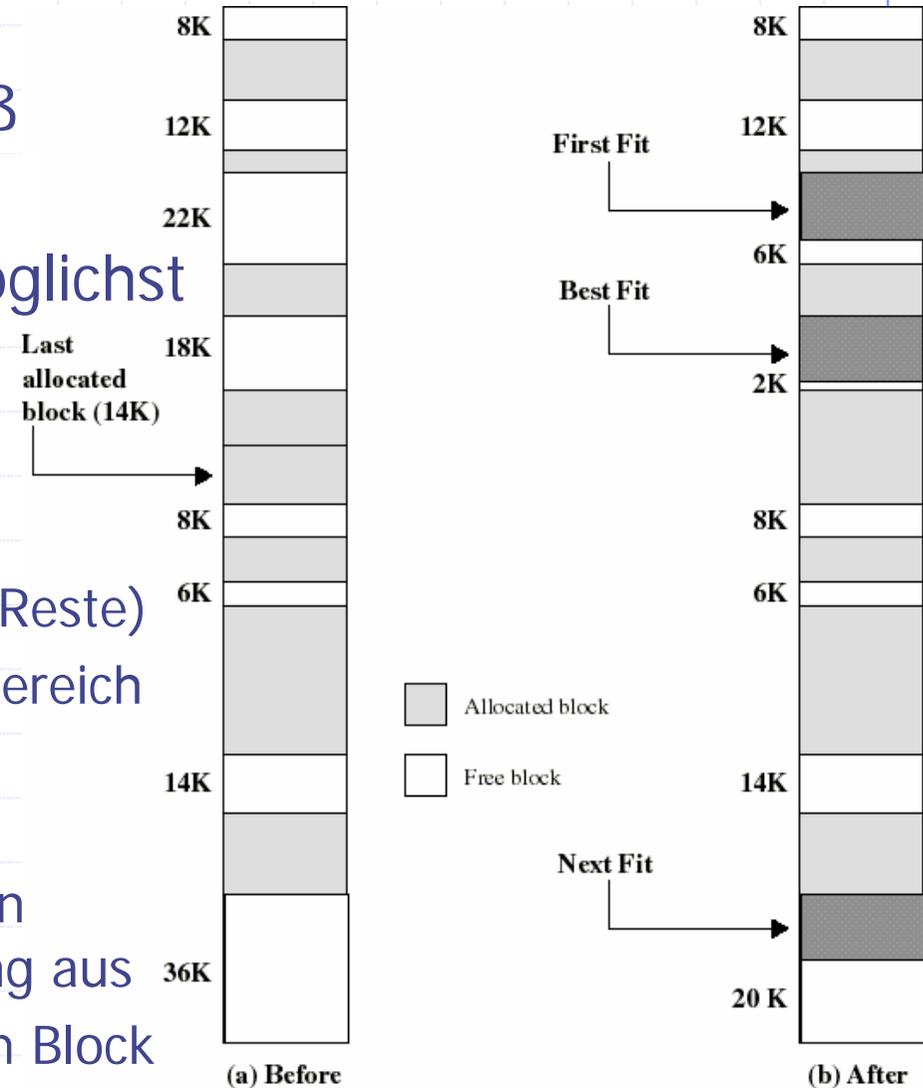
Dynamische Partitioning: Beispiel



- ◆ Alle Prozesse blockieren und OS lagert P1 aus, um P2 wieder einzulagern (Achtung: Relocation von P2)
- ◆ Es bestehen 2 Löcher mit je 96 KB.
- ◆ Kompaktifizierung verschiebt alle Prozesse (nach oben), es entstehen 256 KB freier Speicher am Ende.

Plazierungsalgorithmus

- ◆ bestimmt wohin ein Prozeß eingelagert wird.
- ◆ Ziel: Kompaktifizierung möglichst selten (wg Rechenzeit)
- ◆ Alternativen:
 - **Best-fit:** kleinsten Bereich (liefert viele nutzlose kleine Reste)
 - **First-fit:** ersten passenden Bereich vom Anfang aus (bestes Verfahren)
 - **Next-fit:** nächsten passenden Bereich von letzter Platzierung aus (allokiert schnell aus großem Block am Ende)

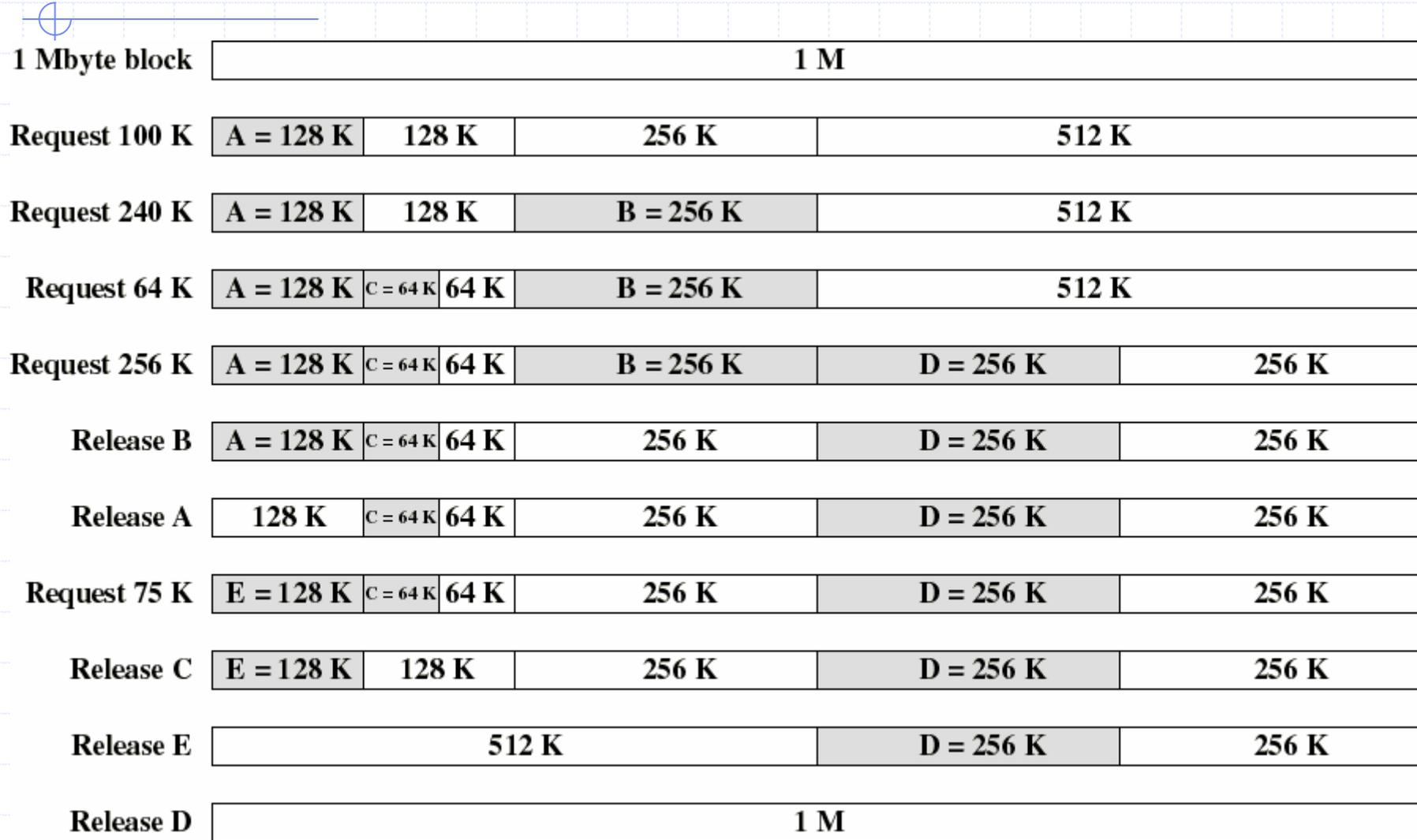


Example Memory Configuration Before and After Allocation of 16 Kbyte Block

Buddy System

- ◆ Brauchbare Kompromißvariante zu fester und variabler Partitionierung, jedoch virtuellem Speicher unterlegen
- ◆ Modifizierte Fassung im Unix SVR4 Kernel
- ◆ interessant für Speicherverwaltung paralleler Programme
- ◆ etwa 25% interne Fragmentierung, jeder Bereich ist mindestens zu 50% genutzt
- ◆ Idee:
 - vergebe Speicherbereiche in Größen als Potenzen von 2, d.h. 2^K wobei $L \leq K \leq U$ und
 - 2^L = kleinste allozierbare Größe eines Bereichs
 - 2^U = größte allozierbare Größe eines Bereichs (generell der komplette verfügbare Speicher)

Beispielablauf im Buddy System



Idee: bei Bedarf Bereich halbieren, nachher Nachbarn verschmelzen

Buddy System: Verfahren

- ◆ Beginne mit kompletten Bereich der Größe 2^U
- ◆ Bei einer Anfrage der Größe S :
 - falls $2^{U-1} < S \leq 2^U$ allokiere Bereich der Größe 2^U
 - sonst, **teile** Bereich in 2 **Buddies** der Größe 2^{U-1} und wiederhole Verfahren mit einem der Buddies.
- ◆ Verfahren endet bei dem kleinsten Bereich $\geq S$
- ◆ 2 **Buddies** werden zum nächstgrößeren Bereich **verschmolzen**, sobald beide frei werden.
 - es existieren auch Varianten bei denen Verschmelzen erst zu einem späteren Zeitpunkt stattfindet (lazy)
- ◆ OS verwaltet freie Bereiche der Größe 2^i in einer Liste i , so daß Verschmelzungsoperationen und Anfragen jeweils an passender Liste ansetzen können.

Bisher einfache Speicherverwaltung ohne virtuellen Speicher

- ◆ Annahme: ausführbare Prozesse werden komplett in den Hauptspeicher geladen
- ◆ folgende Grundlagentechniken wurden betrachtet
 - Feste Partitionierung (fixed partitioning)
 - Dynamische Partitionierung (dynamic partitioning)
 - Buddy System
- ◆ Einschub: unterschiedliche **Adreßarten** wg Änderbarkeit des Speicherortes (Relocation)
- ◆ danach
 - einfaches Seitenverfahren (simple paging)
 - einfaches Segmentierungsverfahren (simple segmentation)

Adreßarten

- ◆ Eine **physikalische Adresse** (absolute Adresse) ist die Hardwareadresse eines Speicherortes im Hauptspeicher
- ◆ Eine **logische Adresse** ist ein Verweis auf einen Speicherort innerhalb eines Programms unabhängig von der physikalischen Struktur des Speichers.
 - Compiler erzeugen Code mit logischen Adressen.
 - Eine **relative Adresse** ist eine logische Adresse, bei der der Speicherort relativ zu einem anderen bekannten Speicherort des Programms angegeben ist, z.B. als sog. Offset zur logischen Anfangsadresse 0 eines Programms.
(Offset = additive Konstante)
- ◆ Adressen werden innerhalb eines Programms als Verweis auf Objekte/Daten und als Zieladresse von Sprüngen verwendet.

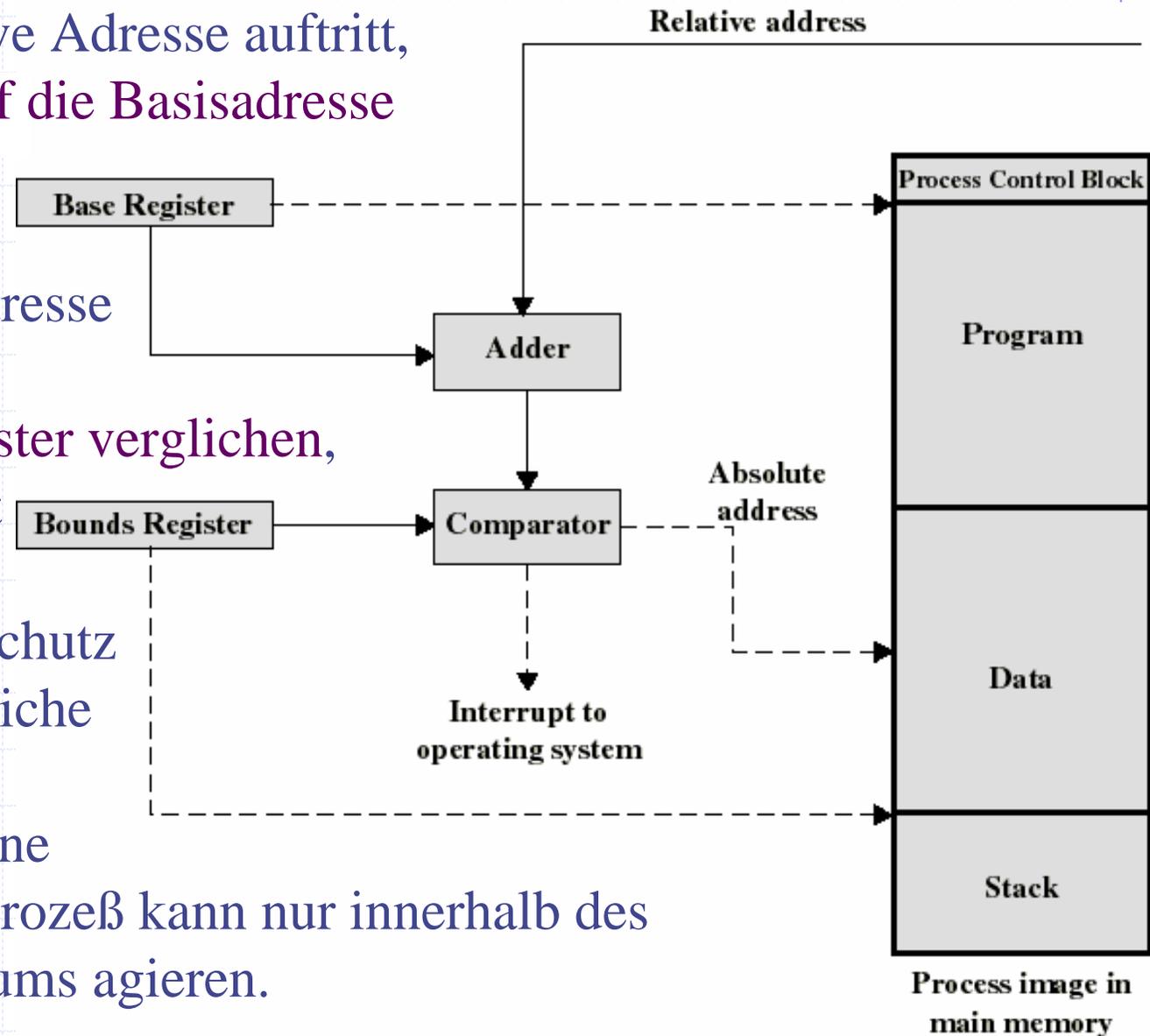
Abbildung logischer->physikalischer Adressen zur Laufzeit

- ◆ Relative Adressen sind häufigster Typ logischer Adressen in ausführbaren Programmen (Modulen).
- ◆ Ein Programm wird mit relativen Adressen in den Speicher geladen.
- ◆ Bei der Ausführung von Instruktionen werden logische Adressen in physikalische Adressen mit passender Hardware umgerechnet, z.B.
 - wenn ein Prozeß in den Zustand **Running** wechselt, wird ein Basis Register der CPU mit der physikalischen Anfangsadresse des Prozeßimages geladen = Wert des Bezugspunkts für die logische Anfangsadresse 0.
 - Ein Begrenzungsregister wird mit der physikalischen Endadresse des Prozeßimages geladen.

Beispiel: Hardware zur Adreßtransformation

Wenn eine relative Adresse auftritt, wird der Wert auf die Basisadresse aufaddiert.

Die resultierende physikalische Adresse wird mit dem Begrenzungsregister verglichen, ggfs ein Interrupt erzeugt. Dadurch wird ein Zugriffschutz für Speicherbereiche anderer Prozesse auf Hardwareebene realisiert. Jeder Prozeß kann nur innerhalb des eigenen Adreßraums agieren.

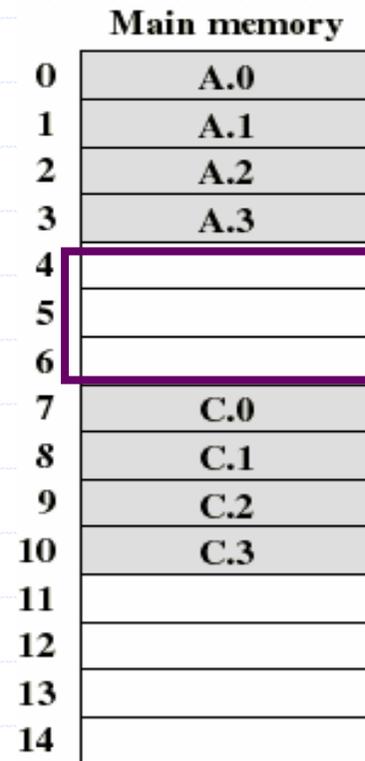


Einfaches Seitenverfahren (Paging)

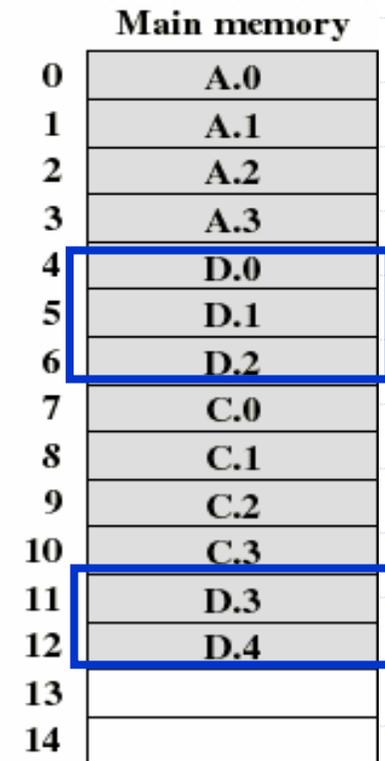
- ◆ Der Hauptspeicher wird in gleichgroße Bereiche (Rahmen, Frames) fester Größe aufgeteilt (wie bei fester Partitionierung, nur Größe relativ gering)
- ◆ Trick: jeder Prozeß wird in Bereiche genau derselben Größe aufgeteilt, sog. Seiten, Pages.
- ◆ Die Seiten jedes Prozesse werden einzeln jeweils Rahmen zugeordnet. Die Zuordnung ist dynamisch.
- ◆ Effekt: ein Prozeßimage erstreckt sich über viele Seiten, belegt aber nicht notwendigerweise ein fortlaufendes Stück Speicher.

Beispiel für Paging

- ◆ Ursprünglich seien Prozesse A,B,C aufsteigend eingelagert worden. Prozeß B wird ausgelagert und D wird eingelagert. D erhält keine fortlaufenden Seiten.
- ◆ Keine externe Fragmentierung!
- ◆ Interne Fragmentierung jeweils nur auf der letzten Seite eines Prozesses.



(e) Swap out B



(f) Load Process D

Seitentabellen

0	0
1	1
2	2
3	3

Process A
page table

0	—
1	—
2	—

Process B
page table

0	7
1	8
2	9
3	10

Process C
page table

0	4
1	5
2	6
3	11
4	12

Process D
page table

13
14

Free frame
list

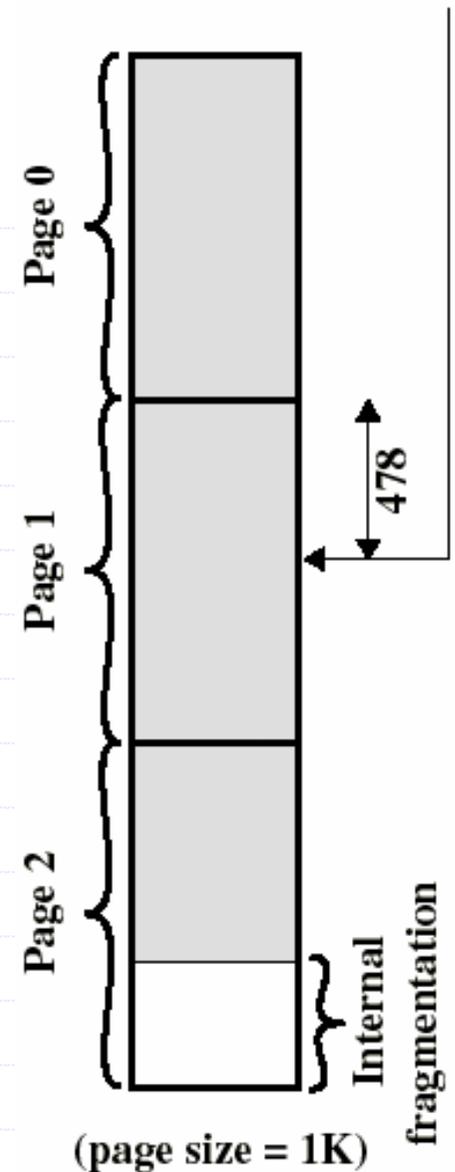
- ◆ Das OS braucht für jeden Prozeß eine **Seitentabelle**, die im Hauptspeicher präsent sein muß, und eine Liste der freien Seitenrahmen.
- ◆ Jeder Eintrag besteht aus einer **Rahmennummer** mit dem entsprechenden physikalischen Speicherort.
- ◆ Die Seitentabelle hat die **Seitennummer** als Index.
- ◆ Paging erfordert **logische Adressen** aus Seitennummer und Offset innerhalb der Seite.
- ◆ Für die Umrechnung hält ein CPU Register die physikalische Anfangsadresse der Seitentabelle des Prozesses.

Logische Adressen bei Paging

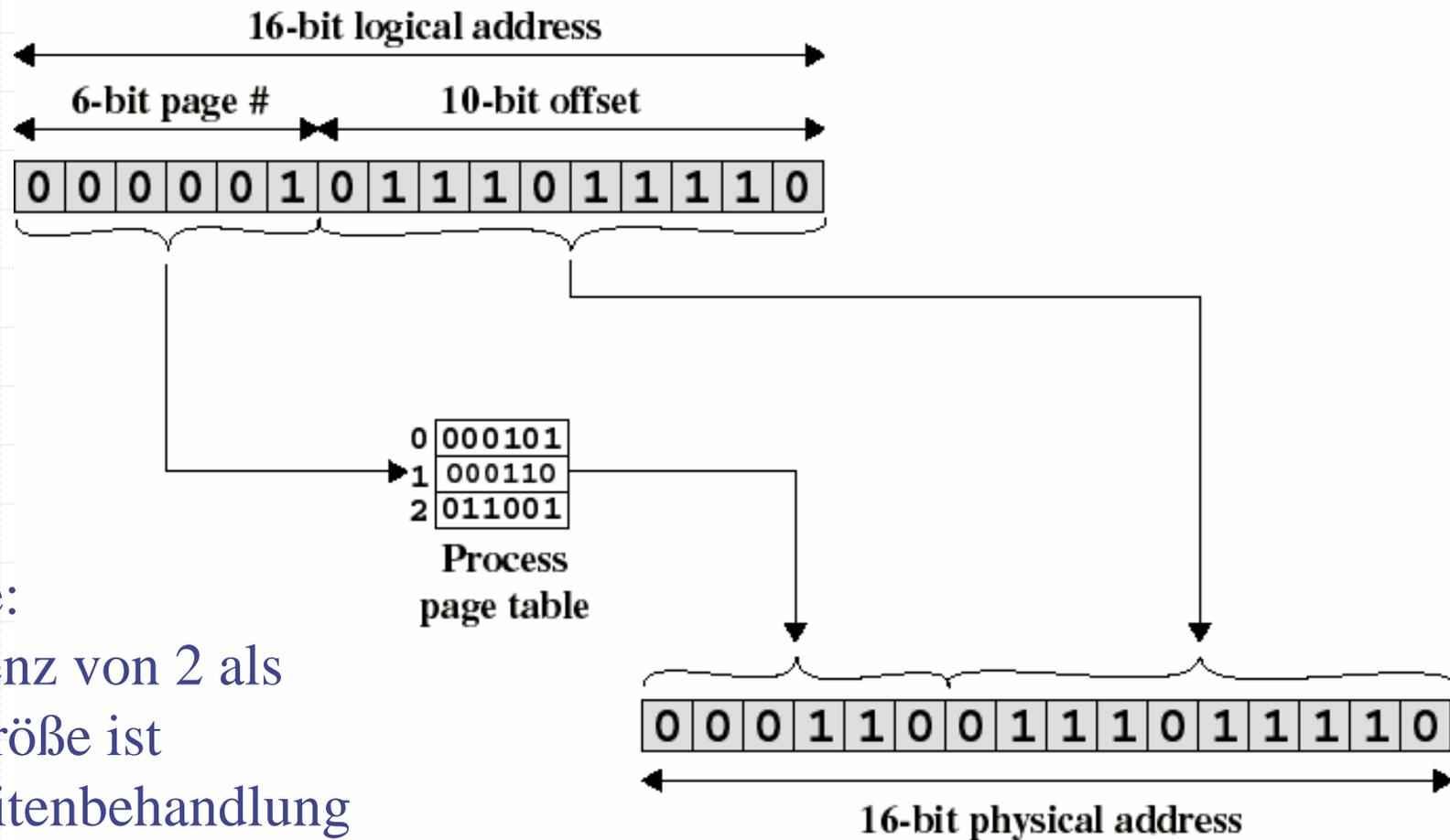
- ◆ Vor.: Seitengröße Potenz von 2
- ◆ Umrechnung aus logischer Adresse in physikalische Adresse:
(Seitennummer, Offset)
-> (Rahmennummer, Offset)
ersetzen, Bitmuster **aneinanderhängen**
- ◆ z.B. 16 Bitadressen, Seitengröße 1KB,
=> 10 bits für den Offset, bleiben 6 Bits für die Seitennummer übrig
- ◆ 16 Bit Adresse aus 10 Bits Offset (niederwertig) und **6 Bits Seitennummer** (höherwertig) ist **relative** Adresse des gesamten Prozesses, nach Ersetzen durch **6 Bits Rahmennummer** aus der Seitentabelle ist es die **physikalische** Adresse.

Logical address =
Page# = 1, Offset = 478

0000010111011110



Übersetzung logischer in physikalische Adressen beim Paging



Vorteile:

bei Potenz von 2 als
Seitengröße ist

- die Seitenbehandlung für Programmierer, Compiler, Assembler, Linker **nicht erkennbar**.
- HW Funktion besonders einfach, weil Addition hier einfacher **Konkatenation** entspricht.

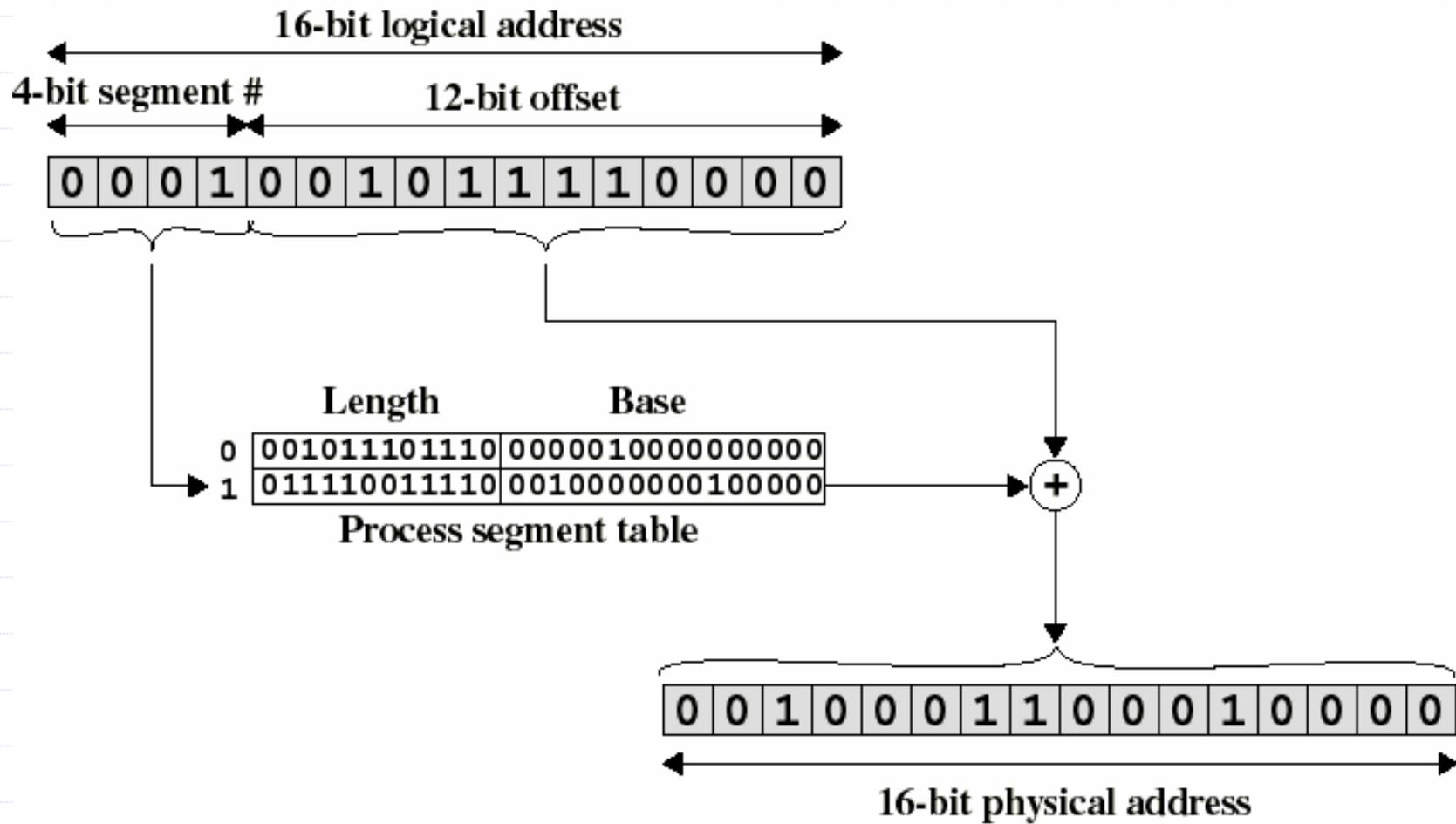
Einfaches Segmentierungsverfahren

- ◆ Jedes Programm wird in Bereiche, **Segmente, ungleicher Größe** geteilt.
- ◆ Wird ein Prozeß geladen, können die Segmente **im Speicher beliebig** verteilt werden.
- ◆ **Keine interne Fragmentierung** wg variabler Segmentgröße
- ◆ **Externe Fragmentierung** tritt auf, kann durch kleine Segmentgrößen reduziert werden.
- ◆ Segmentierung kann vom Programmierer genutzt werden, z.B. Daten in einem Segment, Code in einem anderen.
- ◆ Verwaltung mit **Segmenttabellen** analog zur Seitentabelle, jeweils physikalische Anfangsadresse des Segmentes und die Länge (wg Zugriffsschutz)

Logische Adressen bei Segmentierung

- ◆ Erreicht ein Prozeß den Zustand running, wird ein CPU Register mit der Anfangsadresse der Segmenttabelle des Prozesses gesetzt.
- ◆ Die CPU transformiert eine **logischen Adresse** $(\text{Segmentnummer}, \text{offset}) = (n, m)$, in eine physikalische durch Addition $k+m$, wobei (k, l) der Eintrag aus der n -ten Zeile der Segmenttabelle ist.
 k ist die Anfangsadresse,
 l ist die Länge des Segmentes.
Das Resultat muß $k+m < k+l$ erfüllen, damit die Adresse zulässig ist (wird durch Hardware geprüft).

Adreßübersetzung bei Segmentierung



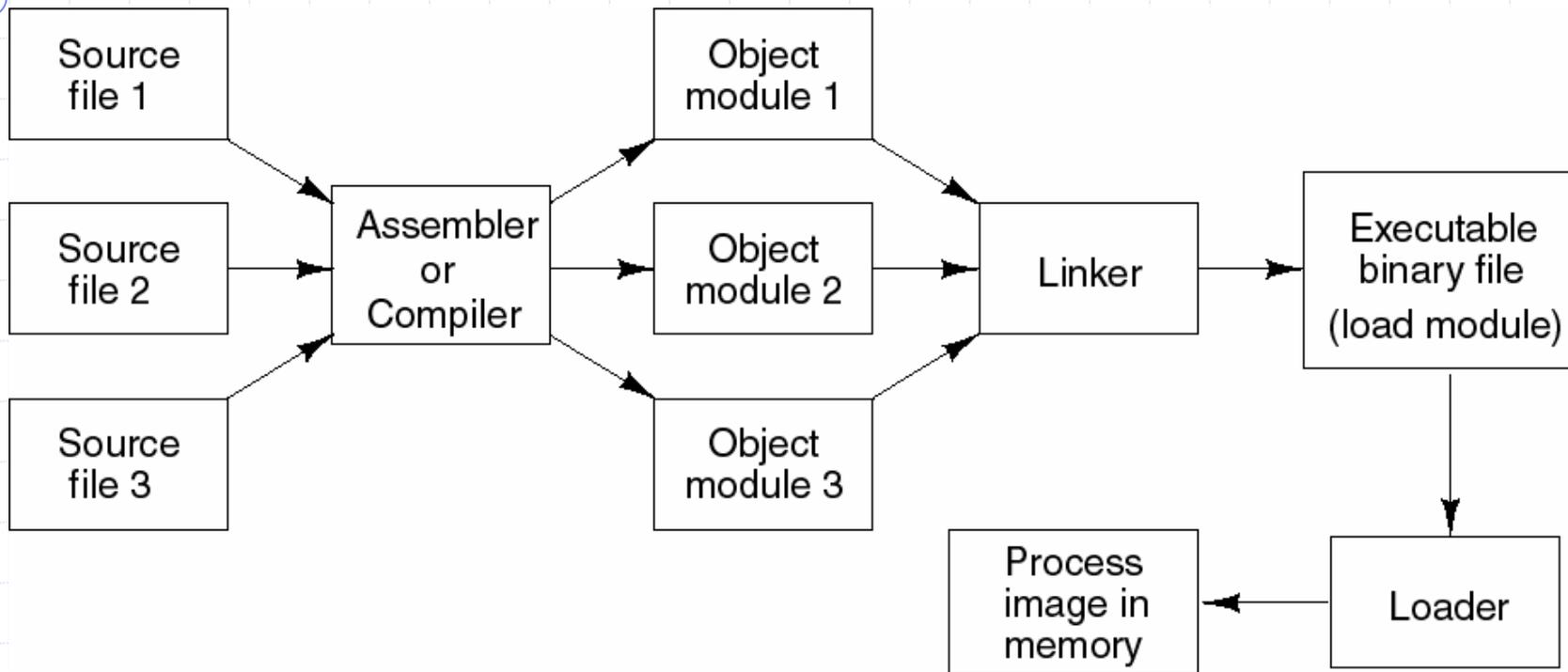
Diskussion einfaches Paging und einfache Segmentierung

- ◆ Segmentierung erfordert mehr Hardware für die Adreßtransformation.
- ◆ Segmentierung hat externe Fragmentierung.
- ◆ Paging hat nur wenig interne Fragmentierung.
- ◆ Segmentierung ist für den Programmierer erkennbar und zur Strukturierung des Programms in Segmente mit unterschiedlichen Zugriffsbeschränkungen (nur ausführbar für Code, lesbar/schreibbar für Daten).
- ◆ Paging ist nicht erkennbar.

Stallings: Kapitel 7, Appendix 7a: Binden und Laden

- ◆ Bisher: einfaches Paging und einfache Segmentierung mit entsprechender Adreßtransformation
- ◆ Frage: wie entsteht eigentlich ausführbarer Code mit logischen (relativen) Adressen aus einem Programm in einer Hochsprache ?
 - Bei einer interpretierten Sprache ?
 - Bei einer compilierbaren Sprache ?
 - Welche Fehlermeldungen entstehen in welchem Schritt ?
- ◆ 1. Schritt: Compiler erstellt zu jedem Quelltext Modul ein Objekt Modul (object file).
- ◆ 2. Schritt: Binder (Linker) verbindet Module zu einem ausführbaren Gesamtprogramm (executable binary).

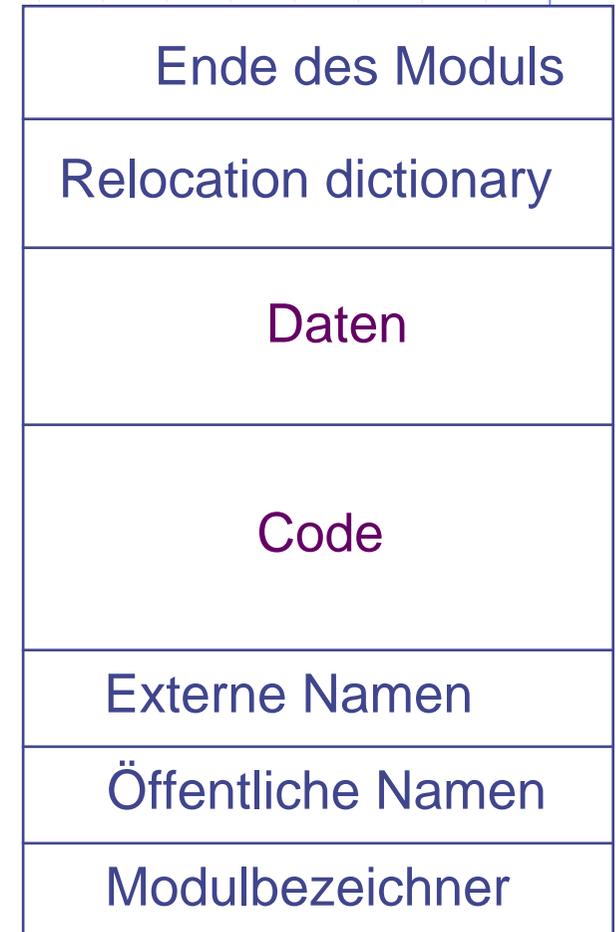
Schritte zum Laden eines Prozesses in den Speicher



- ◆ Der Lader platziert ein ausführbares Programm im physikalischen Speicher.

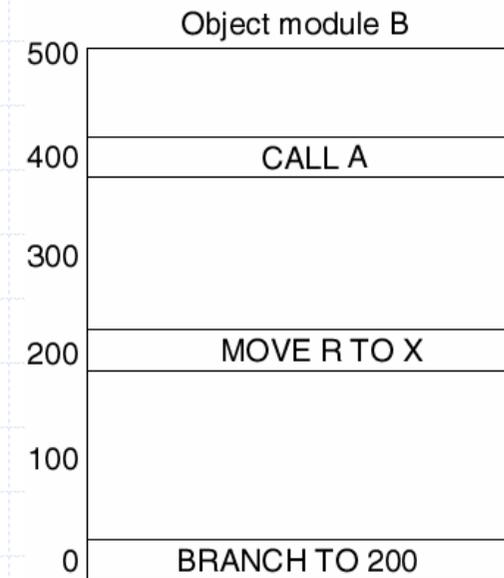
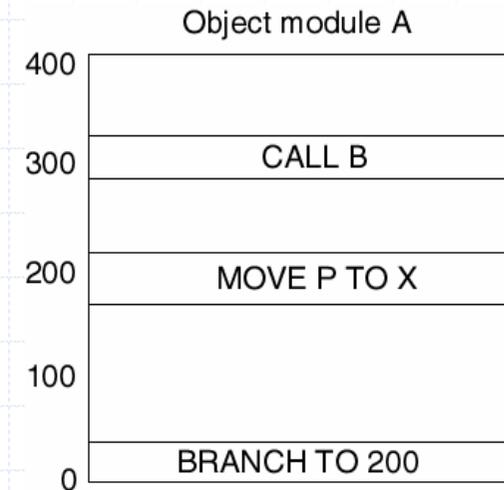
Object Module (keine Segmentierung)

- ◆ Nur **Code** und **Daten** werden in den Speicher geladen, alles andere dient nur dem Binder und wird entfernt.
- ◆ Öffentliche Namen können von anderen Modulen genutzt werden, sind tabelliert.
- ◆ Externe Namen werden in anderen Modulen definiert. Die Tabelle enthält eine Liste der Instruktionen, die diese verwenden.
- ◆ Alle Adressen sind relativ zu einem Bezugspunkt innerhalb des Moduls.
- ◆ Relocation Dictionary enthält eine Liste der Instruktionen, die Verweise als Operanden haben.



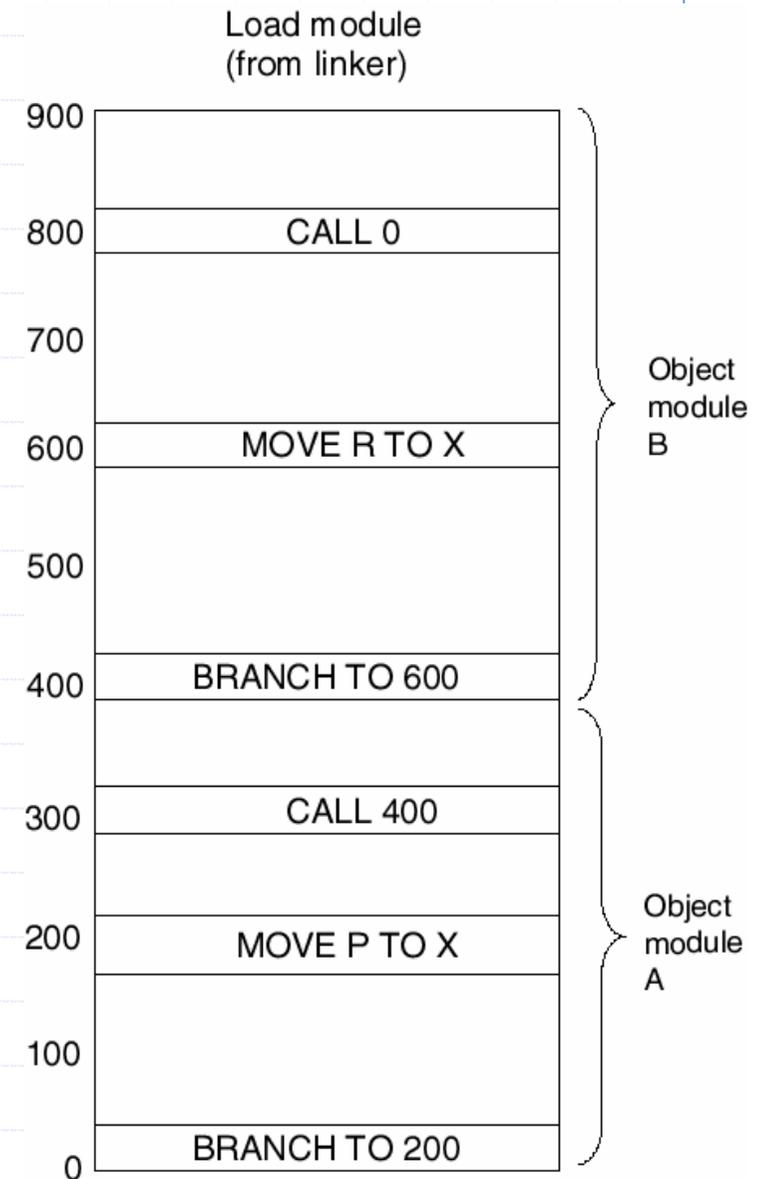
Adreßraum eines Object Moduls

- ◆ Compiler erzeugt jedes Modul mit eigenem Adreßraum. Der Binder ändert dies.
- ◆ Ohne Segmentierung (Abb.->)
 - Alle Module werden zu einem gemeinsamen linearen Adreßraum verbunden
 - Typischer, üblicher Fall
(z.B.: Windows 95/98/NT und Unix auf Pentiums)
- ◆ Mit Segmentierung:
 - jedes einzelne Segment hat eigenen linearen Adreßraum
(z.B.: OS/2 auf Pentiums)



Funktionen des Binders und Laders

- ◆ Der Binder nutzt die Tabellen um einen einzigen linearen Adreßraum herzustellen.
- ◆ Die neuen Adressen sind logische, relative Adressen.
- ◆ Der Lader plaziert das Load Module des Binders im physikalischen Speicher.
- ◆ Physikalische Adressen werden zur Laufzeit berechnet, das Modul ist innerhalb des Speichers verschiebbar



Dynamisches Binden

- ◆ Das Binden externer Module erfolgt nicht im Binder, Tabellen bleiben in reduziertem Umfang erhalten.
 - Windows: externe Module sind .DLLs
 - Unix: externe Module sind .SO Dateien (shared library)
- ◆ Das Load Module enthält Verweise auf externe Module, diese Verweise müssen aufgelöst werden:
 - beim Laden des Programms (load-time dynamic linking)
 - zur Laufzeit: bei Aufruf einer Methode aus einem externen Modul (run-time dynamic linking)
- ◆ in beiden Fällen muß das OS die externen Module finden und anbinden

Vorteile des dynamischen Bindens

- ◆ Externe Module sind oft OS Hilfsprogramme. Ausführbare Programme können von aktualisierten externen Modulen profitieren ohne neu kompiliert zu werden.
- ◆ Code Sharing: die gleichen externen Module brauchen nur einmal geladen zu werden. Jeder Prozeß wird gegen dasselbe Modul gelinkt.
 - spart Hauptspeicher- und Plattenplatz!
 - wird insbesondere für Bibliotheken genutzt,
Randeffect:
der Speicherbedarf eines Prozeßimages ist nur noch für nichtgeteilte Module konkret ermittelbar

Bemerkung: beliebte Fehlerquelle bei der Portierung / Installation von Software auf „anderen“ Rechner

- Bibliotheken fehlen, werden nicht gefunden (andere Directory Organisation, andere Einstellungen wo Linker/Binder suchen soll)
- Bibliotheken liegen in älteren / neueren Versionen vor.

Zusammenfassung

- ◆ einfache Speicherverwaltung ohne virtuellen Speicher
- ◆ Annahme: ausführbare Prozesse werden komplett in den Hauptspeicher geladen
- ◆ Grundlagentechniken
 - Feste Partitionierung (fixed partitioning)
 - Dynamische Partitionierung (dynamic partitioning)
 - Buddy System
 - einfaches Seitenverfahren (simple paging)
 - einfaches Segmentierungsverfahren (simple segmentation)
- ◆ Vorgang beim Binden und Laden eines Programms
- ◆ beim nächsten Mal:
Speicherverwaltung mit virtueller Speicher