

Betriebssysteme und Netzwerke

Vorlesung 9

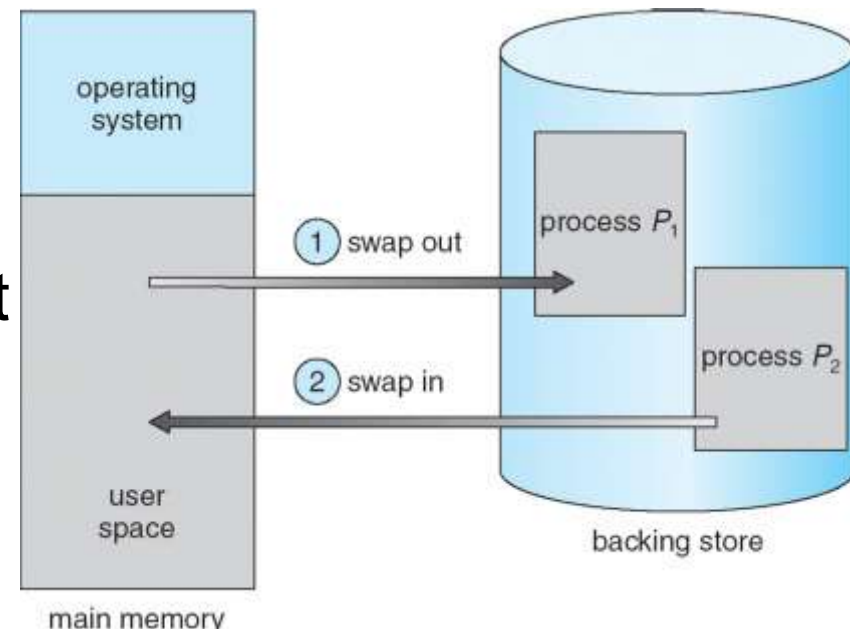
Artur Andrzejak

Umfragen: <https://pingo.coactum.de/301541>

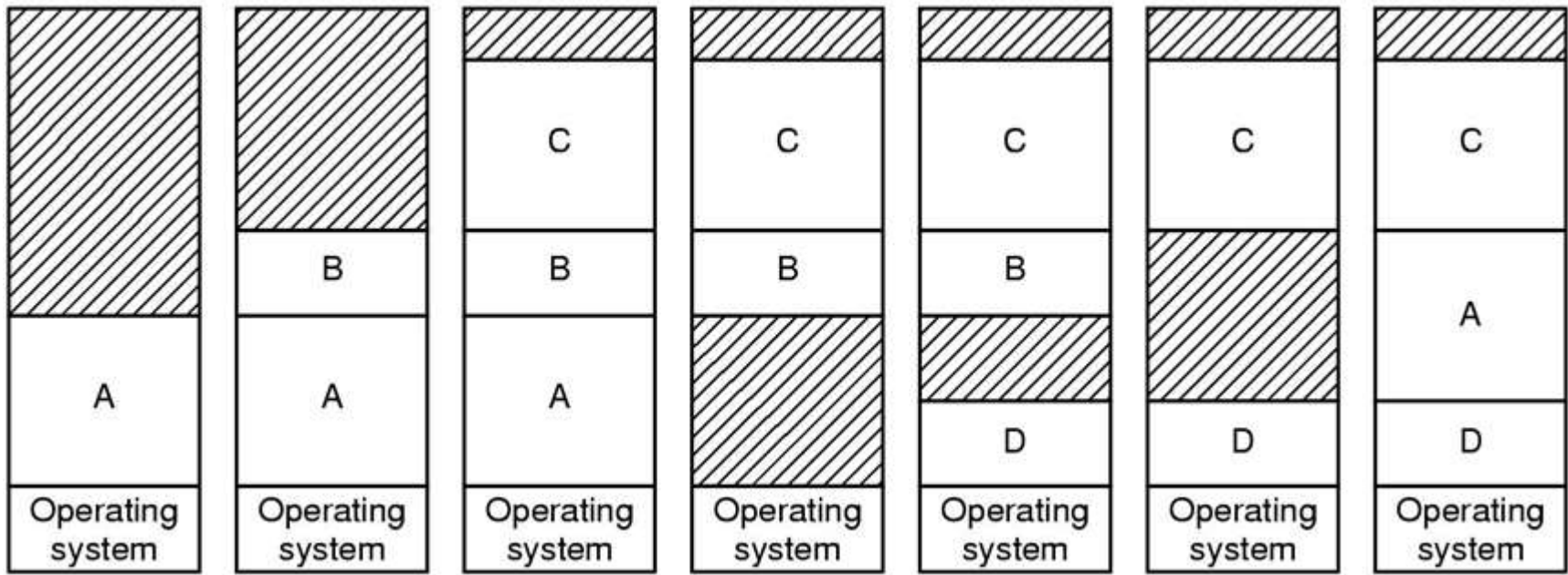
Speichermanagement: Swapping

Swapping - Einführung

- ▶ Hauptspeicher könnte nie groß genug sein
 - ▶ Typisch sind 40-100 Prozesse, jeder 10 MB und mehr
 - ▶ Größere Anwendungen benötigen 100-300+ MB
- ▶ **Swapping**: man lagert das gesamte Speicherabbild eines Prozesses auf die FP aus, um RAM zu schonen
- ▶ **Roll out, roll in** - Prinzip:
Prozesse mit niedriger Priorität werden ausgelagert (swapped out), Prozesse mit höherer hereingeholt
- ▶ In modifizierter Version in UNIX, Linux, Windows



Swapping - Beispiel



Zeit

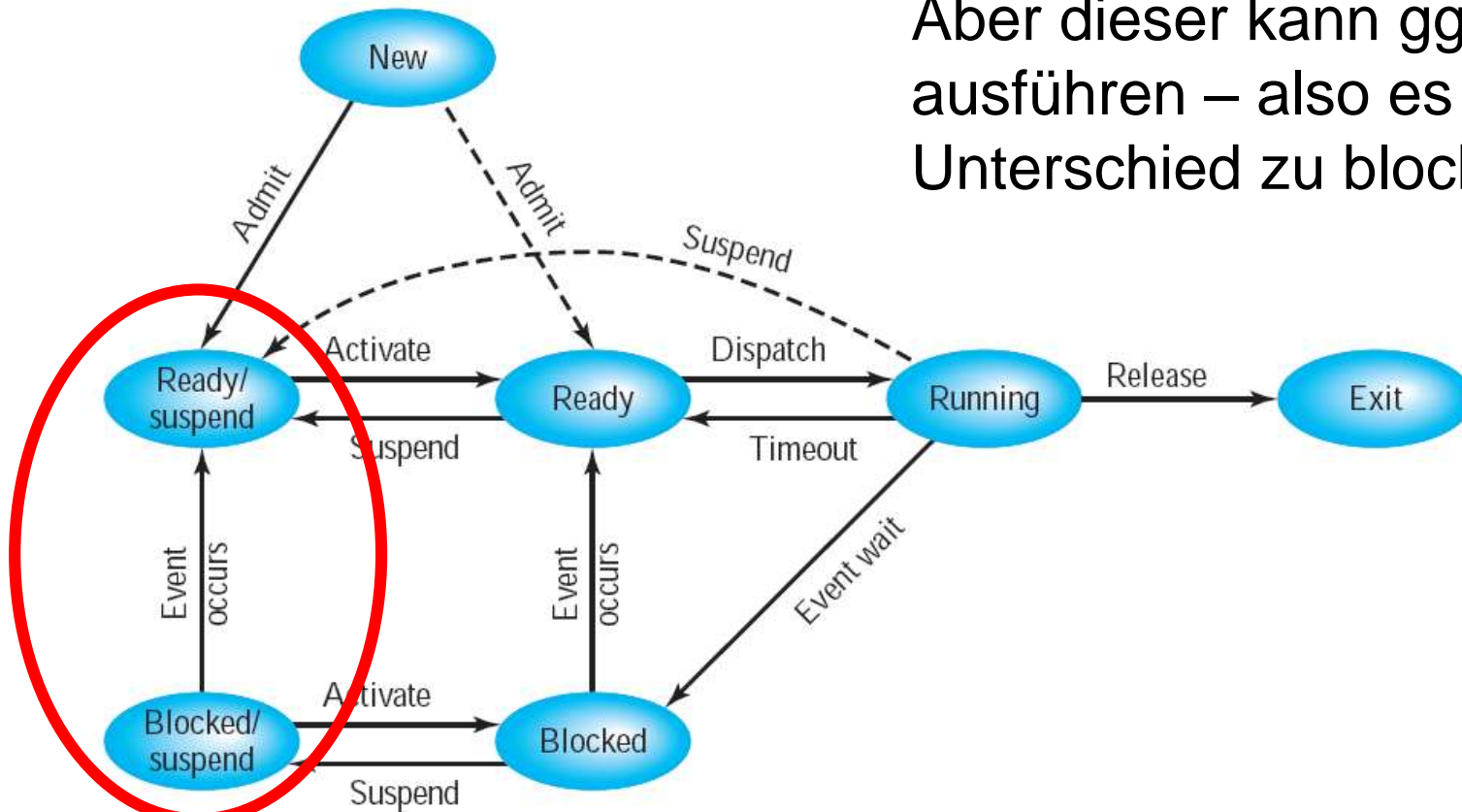
► Probleme:

- Zeit des Datentransfers (z.B. 100 MB Prozess ~ 2+2 Sec.)
- Prozesse, die gerade I/O machen, dürfen nicht ausgelagert werden

Swapping und Prozessscheduling

- Swapping macht das Scheduling der Prozesse etwas schwieriger: ein Prozess, der ausgelagert (**suspended**) ist, wird behandelt als blockiert (d.h. auf I/O-Operation wartend)

Aber dieser kann ggf. ausführen – also es gibt einen Unterschied zu blockiert

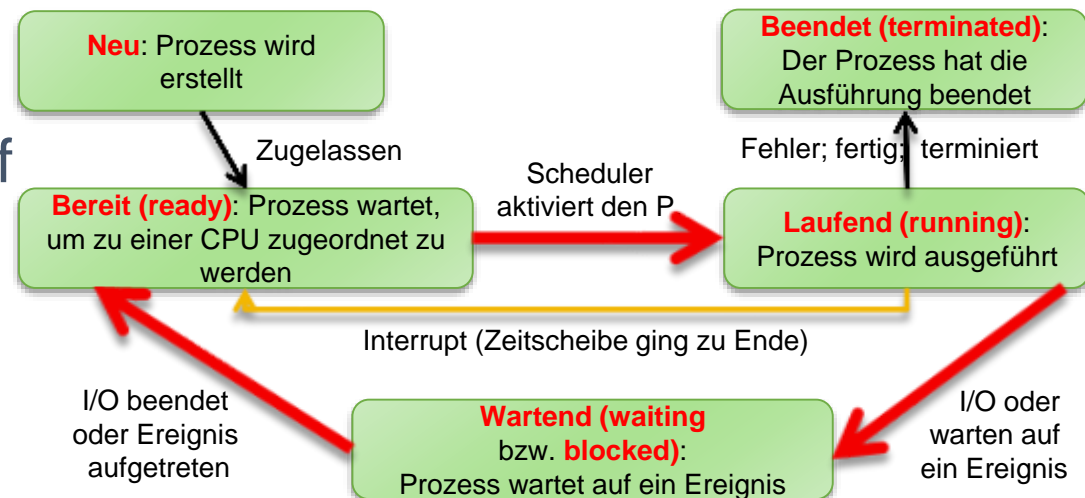


Swapping und Prozessverwaltung

- ▶ Man lagert zunächst solche Prozesse aus, die auf ein Ereignis warten und blockiert sind
 - ▶ Wenn ein ausgelagerter Prozess in den Hauptspeicher zurückgebracht wird, sollte dieser nicht mehr blockiert sein
- ▶ Also reichen die 3 (Haupt-)Prozesszustände **ready**, **running**, **waiting (blocked)** nicht mehr aus!
- ▶ Wir müssen unterscheiden:

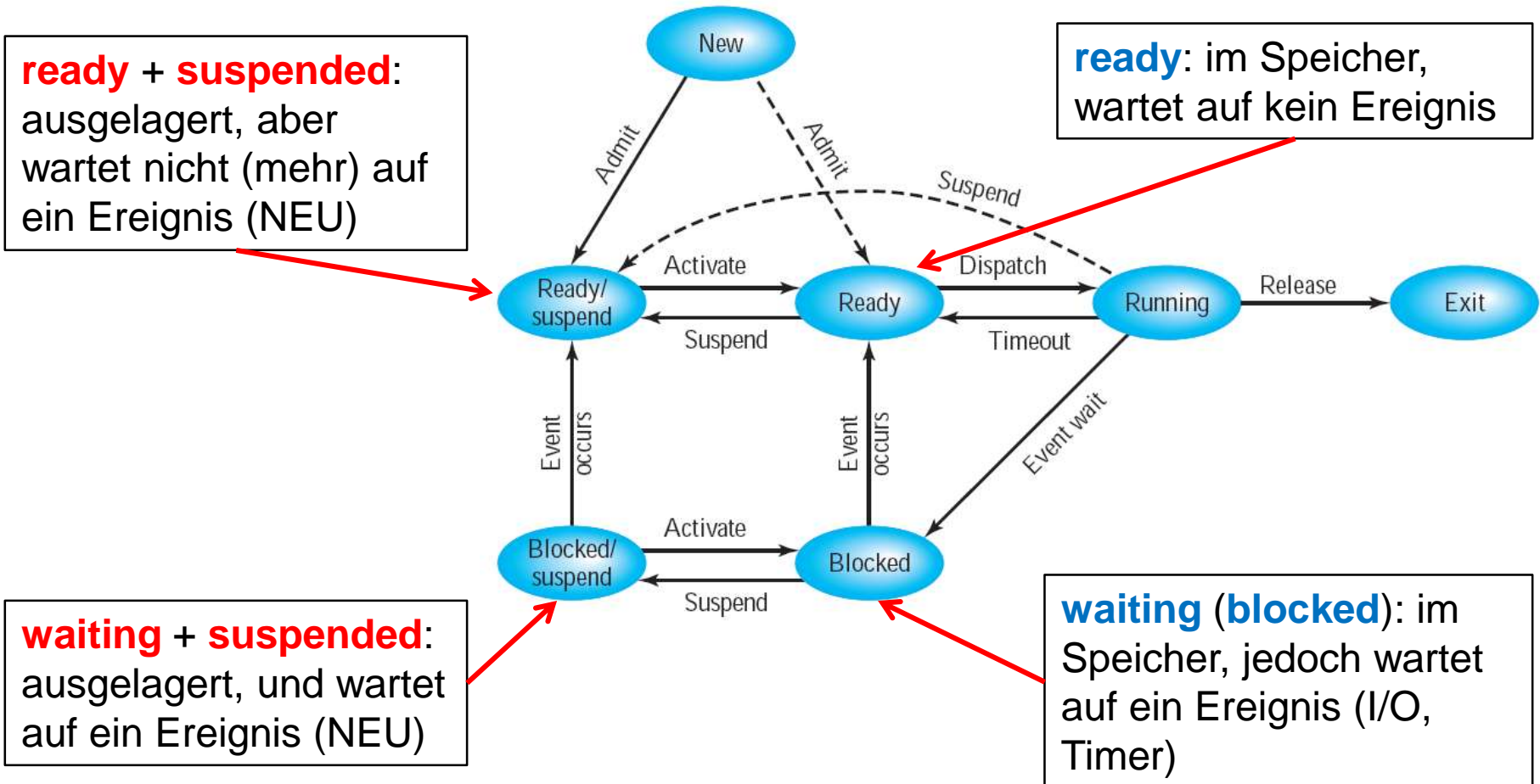
- ▶ **waiting / blocked**: Prozess kann nicht ausführen, weil er auf Ereignis (I/O, ...) wartet

- ▶ **suspended**: P. kann nicht ausführen, weil er ausgelagert ist



Neue Prozesszustände

- Man erhält folgende Kombinationen



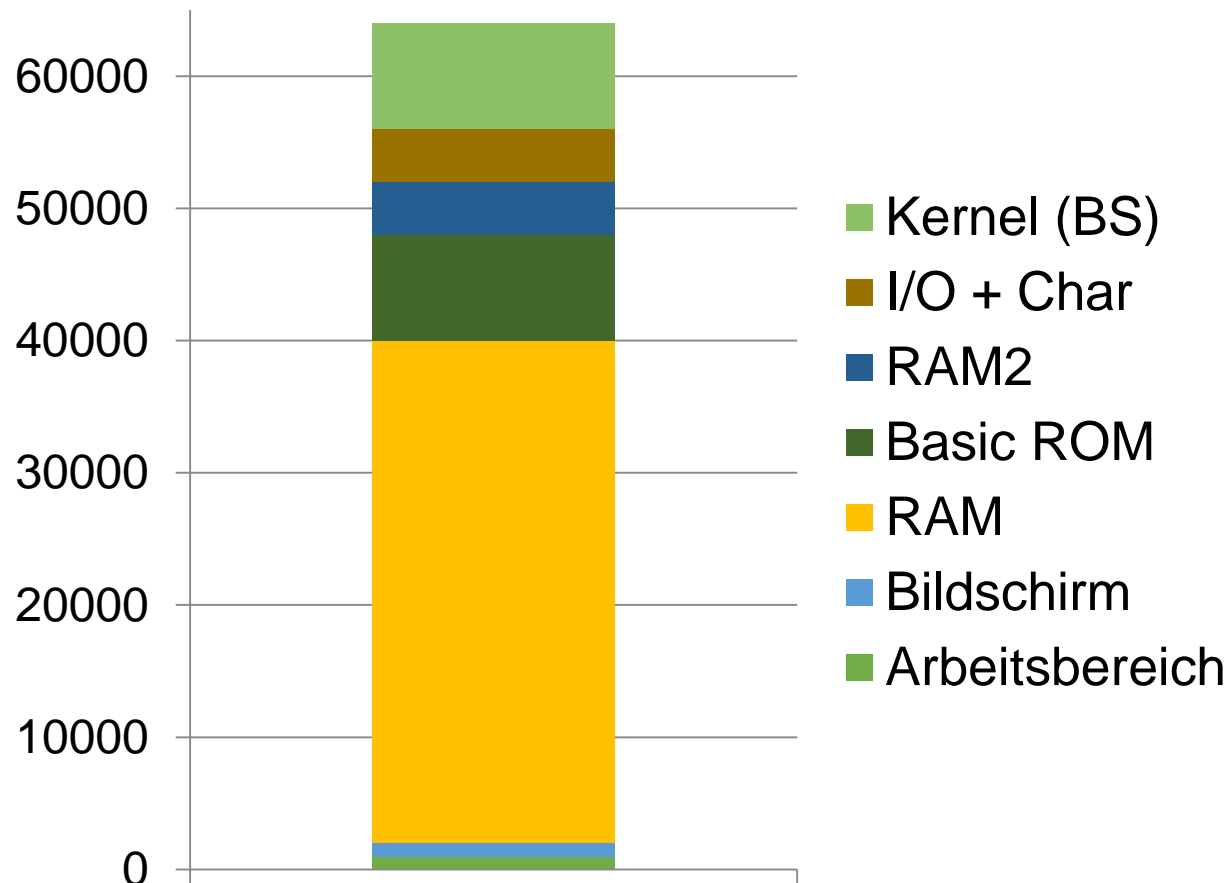
Herausforderungen des Managements von Speicher

Systeme ohne Speicherabstraktion

- ▶ Frühe Systeme hatten keine **Speicherabstraktion**
 - ▶ D.h. die von dem Programm generierte Adresse war zugleich die physische Adresse im Speicher
 - ▶ „Früh“ heißt: Großrechner vor 1960, Minicomputer vor 1970, PCs vor 1980
- ▶ Probleme?
- ▶ Es ist sehr schwierig, mehrere Programme zugleich laufen zu lassen
- ▶ Sehr inflexibel für Änderungen
 - ▶ Ein „längeres“ BS oder weniger / mehr Speicher führen zur Inkompatibilität

Systeme ohne Speicherabstraktion

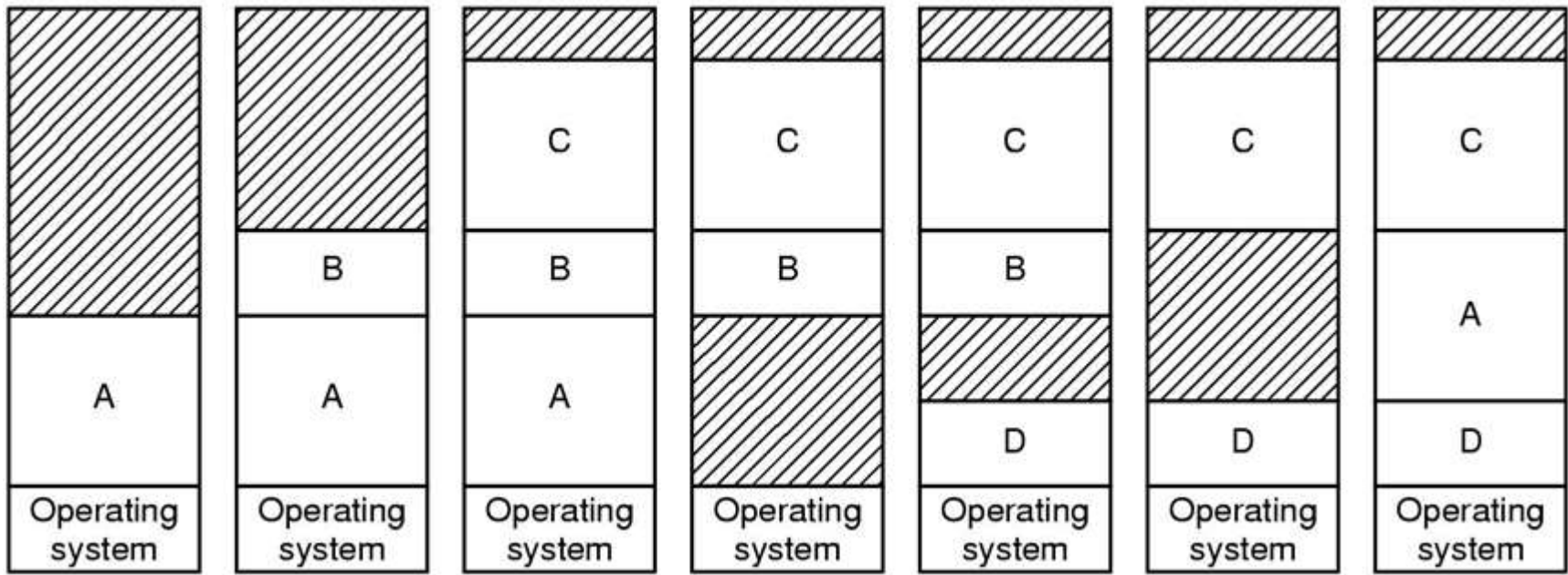
- ▶ Commodore 64 hatte 64 kByte Speicher
 - ▶ Adressraum: auch 64 kByte



Speicherverwaltung

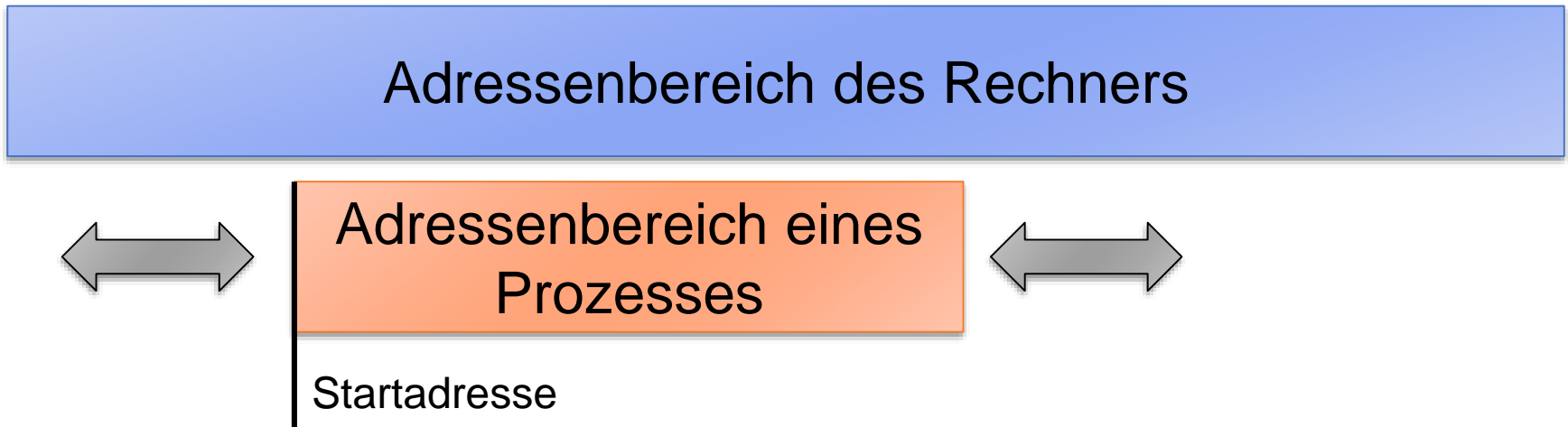
- ▶ Speicheradressen sind zusammenhängend
 - ▶ Gilt für (fast) alle Computersysteme
- ▶ Wie teilen wir den Speicher unter mehreren Prozessen auf?
- ▶ A. Frühe BS teilten jedem Prozess ein Slot von fixer Größe zu (**fixed-sized partitions**)
 - ▶ Z.B. das IBM OS/360
- ▶ B. Alle späteren BS erlaubten variable Partitionsgröße (**variable-partition scheme**)
- ▶ Welche Probleme gibt es in jedem dieser Fälle?

B. Variable Partitionsgröße und Folgen



- ▶ Die variable Partitionsgröße (B) ist die bevorzugte Lösung, hat aber Folgen:
- ▶ 1. Die Startadresse für Code, Daten kann beliebig sein
- ▶ 2. Größe des Adressenbereich eines Prozesses kann variieren (z.B. je nach den Startparametern)

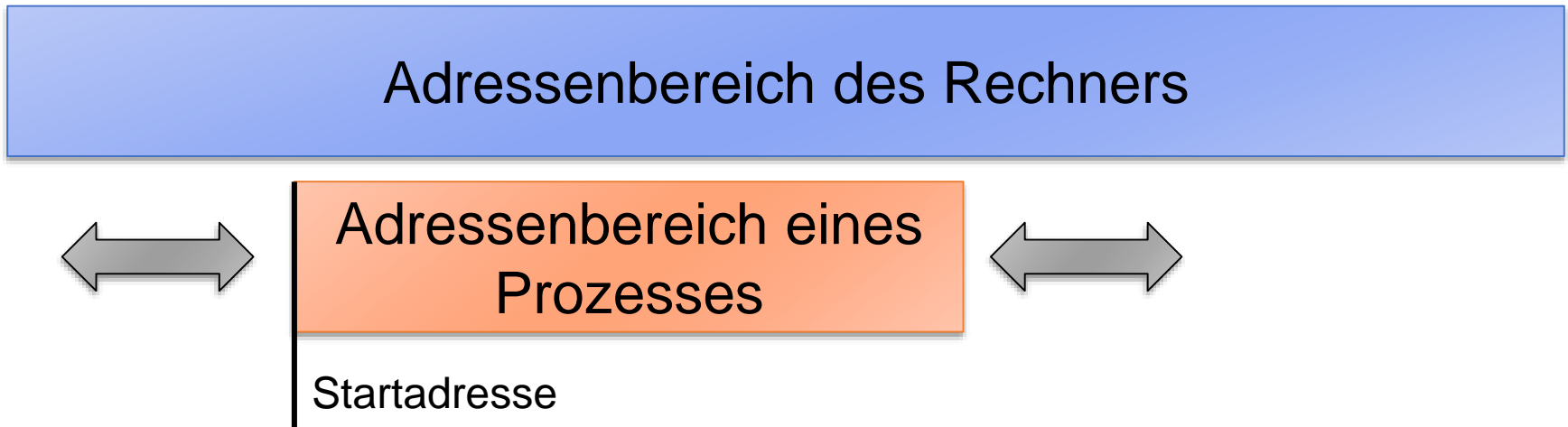
B. Zwei Herausforderungen /1



Das erzeugt zwei technische Herausforderungen:

- ▶ **P1.** Der Code muss ausführen, egal wo die Startadresse (von Code+Daten) liegt
 - ▶ Problem der **Adressenrelokation**
- ▶ **P2.** Adressenbereich muss zusammenhängend sein
 - ▶ Problem der **zusammenhängenden Adressen**

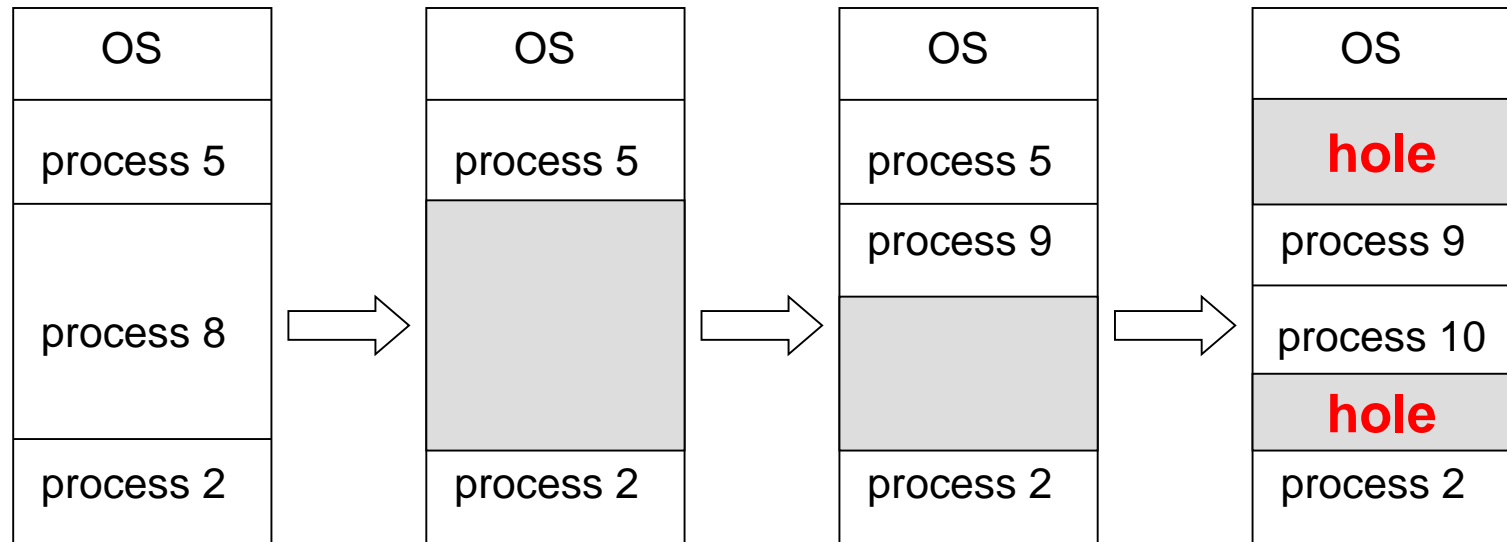
B. Zwei Herausforderungen /2



- ▶ P1. Problem der **Adressenrelokation**
 - ▶ Das ist relativ leicht zu lösen
- ▶ P2. Problem der **zusammenhängenden Adressen**
 - ▶ Die Lösung ist sehr aufwändig, und bedarf Hardware-Unterstützung

P2: Zusammenhängender Speicher

- ▶ Was führt dazu, dass der Adressenbereich (des freien Speichers) ggf. nicht zusammenhängend wird?
- ▶ Typischer Ablauf bei **variable-partition scheme** (B)
 - ▶ Anfangs gibt es nur einen großen Block des Speichers
 - ▶ Im Laufe der Ausführung des BS entstehen mehrere **Lücken (holes)**: Bereiche mit zu wenig Speicher



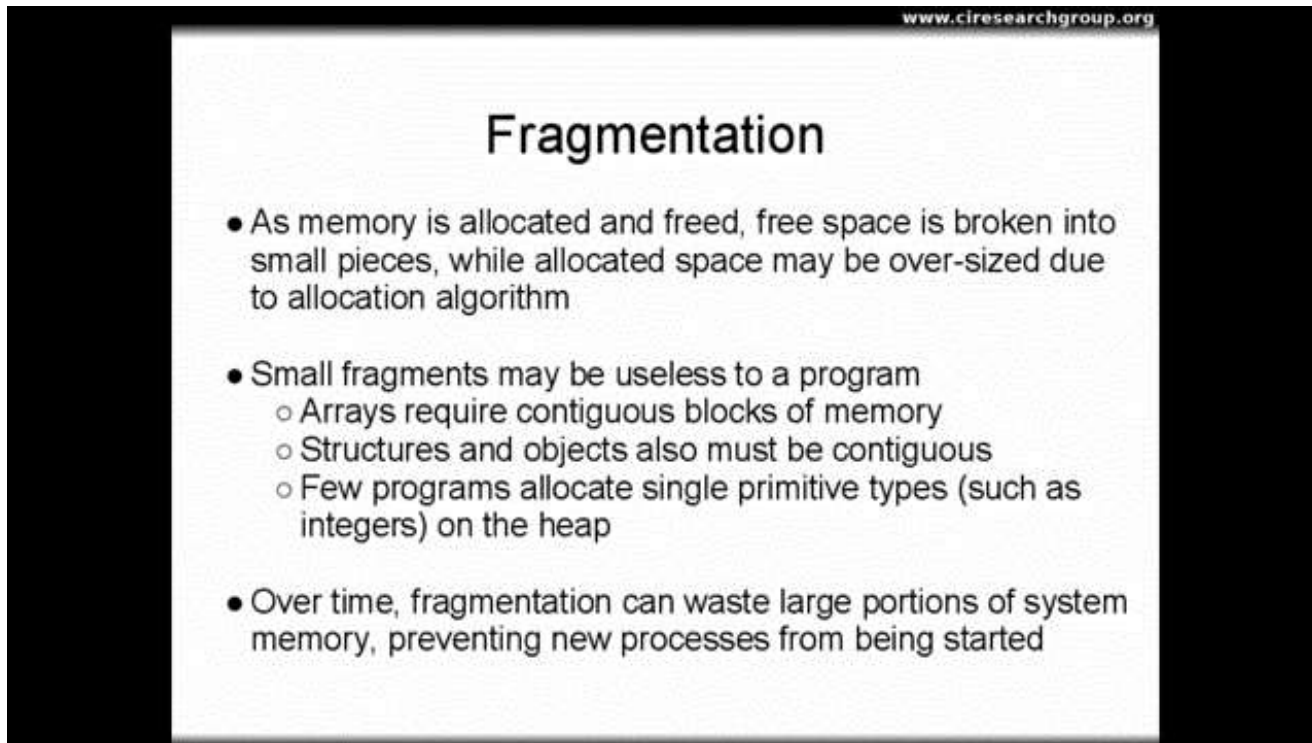
Interne vs. Externe Fragmentierung

- ▶ Diese „Löcher“ verursachen große Ineffizienz
 - ▶ Man endet mit einer Menge von kleinen Löcher, in die kein Prozess mehr passt
 - ▶ Verschwendung von über 30% des Speichers möglich!
- ▶ Das Problem der Löcher nennt man **externe Fragmentierung** (**external fragmentation**)
- ▶ Aber: Bei Slots einer fixen Größe (Schema A) werden die „Endstücke“ der Slots ggf. nicht ausgenutzt
 - ▶ Z.B. ein Prozess mit 5 kB braucht ein 8kB Slot => 3 kB „verschwendet“
 - ▶ Man nennt das **interne Fragmentierung**

Video Externe vs. Interne Fragmentierung

► Dynamic Memory Allocation [08b]

- <https://www.youtube.com/watch?v=Dml54J3Kwm4>
- Ab 2:35 bis ca. 3:52 (min:sec)



www.ciresearchgroup.org

Fragmentation

- As memory is allocated and freed, free space is broken into small pieces, while allocated space may be over-sized due to allocation algorithm
- Small fragments may be useless to a program
 - Arrays require contiguous blocks of memory
 - Structures and objects also must be contiguous
 - Few programs allocate single primitive types (such as integers) on the heap
- Over time, fragmentation can waste large portions of system memory, preventing new processes from being started

Externe Fragmentierung - Abhilfen

- ▶ Externe Fragmentierung ist ein ernsthaftes Problem
- ▶ Lösungen?
- ▶ Das BS kann dann versuchen, alle Prozesse „nach unten“ (zu kleineren Adressen) im Speicher zu verschieben - **Verdichtung (memory compaction)**
 - ▶ Zeit: 1 GB, 4 Bytes werden in 20 ns kopiert => 5 s!
 - ▶ Geht nur bei der Code-Relokation zur Laufzeit (später)
 - ▶ Vorsicht mit I/O!
- ▶ Alternative: Gibt es Strategien für die Platzierung der Prozesse, so dass keine Lücken entstehen?

Platzierung der Prozesse

- ▶ Wo landet ein neuer oder ein wieder eingelagerter o. „zurück-geswappter“ Prozess?
- ▶ Mehrere Möglichkeiten
 - ▶ **First fit**: belege die 1. Lücke, die groß genug ist
 - ▶ **Best fit**: belege die kleinste Lücke, in die der „zurück-geswappte“ Prozess passt
 - ▶ Das hinterlässt die kleinste Lücke
 - ▶ **Worst fit**: belege die größte Lücke
 - ▶ Das hinterlässt die größte Lücke
 - ▶ Generell schlechter als die beiden anderen Ansätze
- ▶ Dazu im Video “Dynamic Memory Allocation” ([Link](#)) – siehe 3:52 bis 5:40++

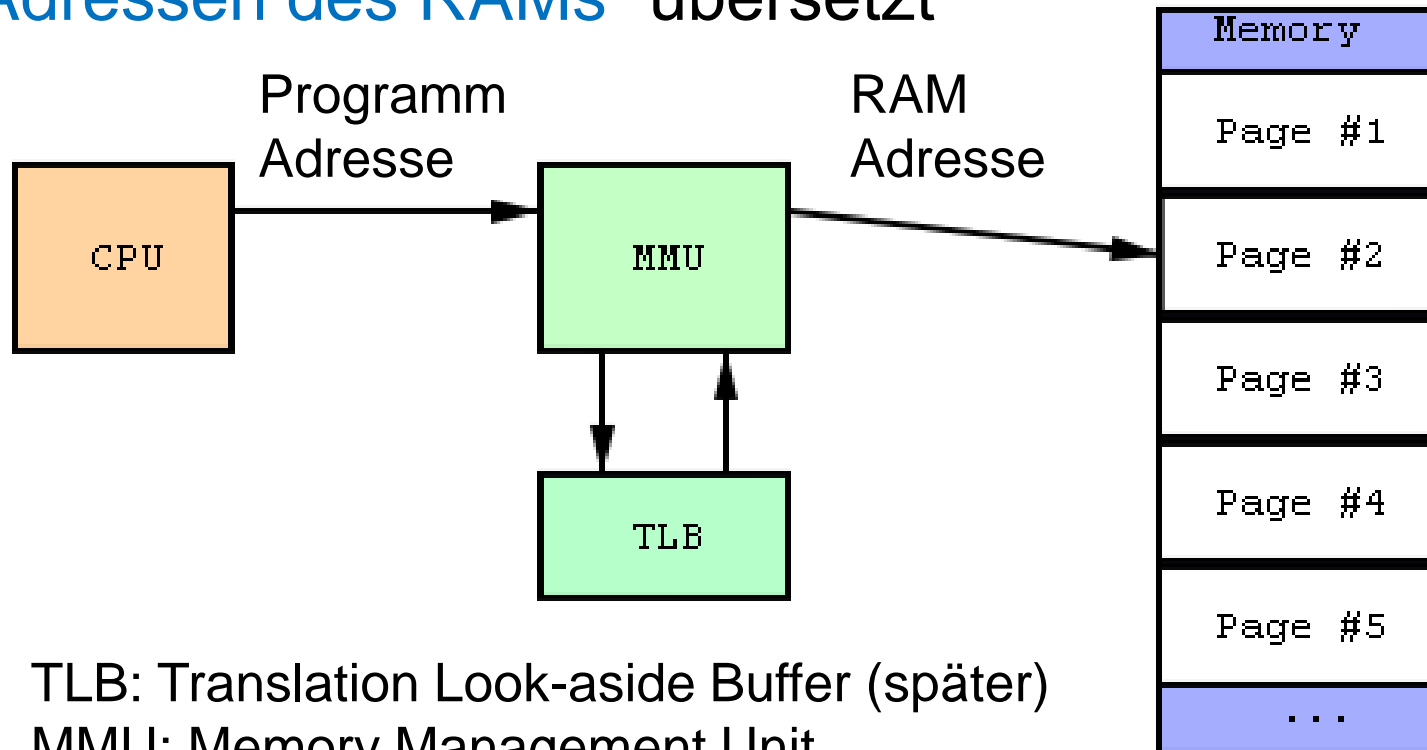
Problem der Adressenrelokation: Lösungen

Relokation bzw. Adressanpassung

- ▶ Bei Multiprogramm-BS kann die Startadresse von Code + Daten **an einer beliebigen Stelle im physischen Speicher** liegen
 - ▶ Normalerweise erwartet der Code, dass die Adressen in Speicher aus seiner Sicht bei der Adresse 0 anfangen
- ▶ Die Lösungen dazu nennt die **Adressanpassung** bzw. **Relokation (address binding)**
- ▶ Es gibt mehrere Lösungen
 - ▶ Abhängig von der Phase der Programm-Entwicklung bzw. -Ausführung

Memory-Management-Unit (MMU)

- ▶ Zentrales Element der Lösung ist ein **Memory-Management Unit** - eine Hardwareeinheit (oft im Prozessor), die die „Adressen des Programms“ auf die „Adressen des RAMs“ übersetzt



TLB: Translation Look-aside Buffer (später)
MMU: Memory Management Unit

Logischer vs. Physischer Adressraum

- ▶ MMU übersetzt den **logischen Adressraum** auf den unabhängigen **physischen Adressraum**
- ▶ Logischer Adressraum besteht aus **logischen** oder **virtuellen Adressen**
 - ▶ Der Prozess und die CPU „sehen“ nur diese logische Adressen
 - ▶ Werte in den CPU-Registern sind alle logisch
- ▶ PA besteht aus den **physischen Adressen**
 - ▶ Adressen, die von dem Speicher(bus) gesehen wird, d.h. „echte“ oder Hardware-Adressen
 - ▶ Adressen, die die RAM-Bausteine tatsächlich sehen

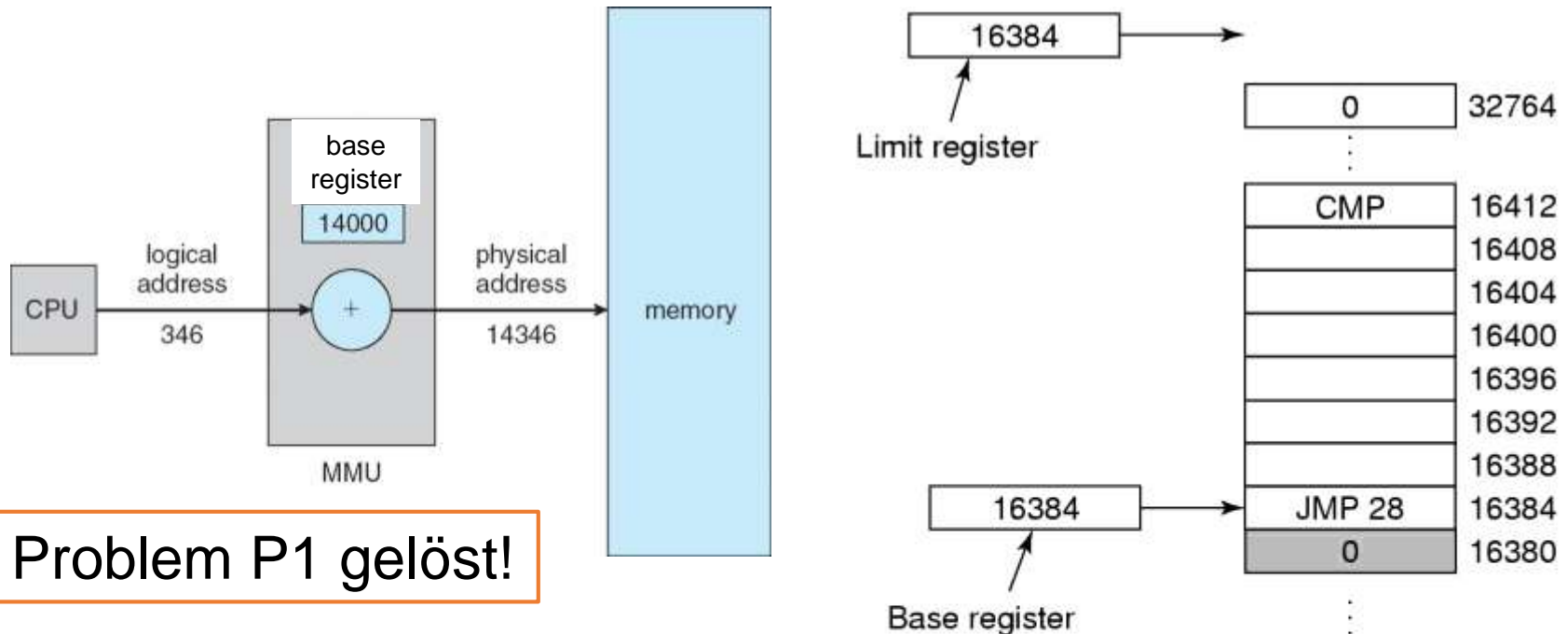
Basis und Limitregister

- ▶ Im einfachsten Fall besteht MMU aus einem **Basis-** und einem **Limitregister**
- ▶ **Basisregister** enthält den Anfang des physischen Adressraumes eines Prozesses
 - ▶ Eine physische Adresse
- ▶ **Limitregister** enthält das Ende des logischen Adressraumes des Prozesses
 - ▶ Eine logische Adresse (ggf. +1)
 - ▶ Normalerweise: Größe des (logischen) Adressraumes
- ▶ Was muss MMU bei einem Speicherzugriff machen (d.h. wie läuft die Adressübersetzung ab)?

Basis und Limitregister /2

- ▶ Wenn eine Adresse referenziert wird ...
 1. Wird es überprüft, ob die **logische Adresse kleiner als der Inhalt des Limitregisters** ist
 2. Der **Inhalt des Basisregisters wird zu der logischen Adresse hinzuaddiert** => physische Adresse
 3. Es wird auf den Speicherinhalt mit der berechneten Adresse (== physische Adresse) zugegriffen

Memory-Management-Unit (MMU)



Problem P1 gelöst!

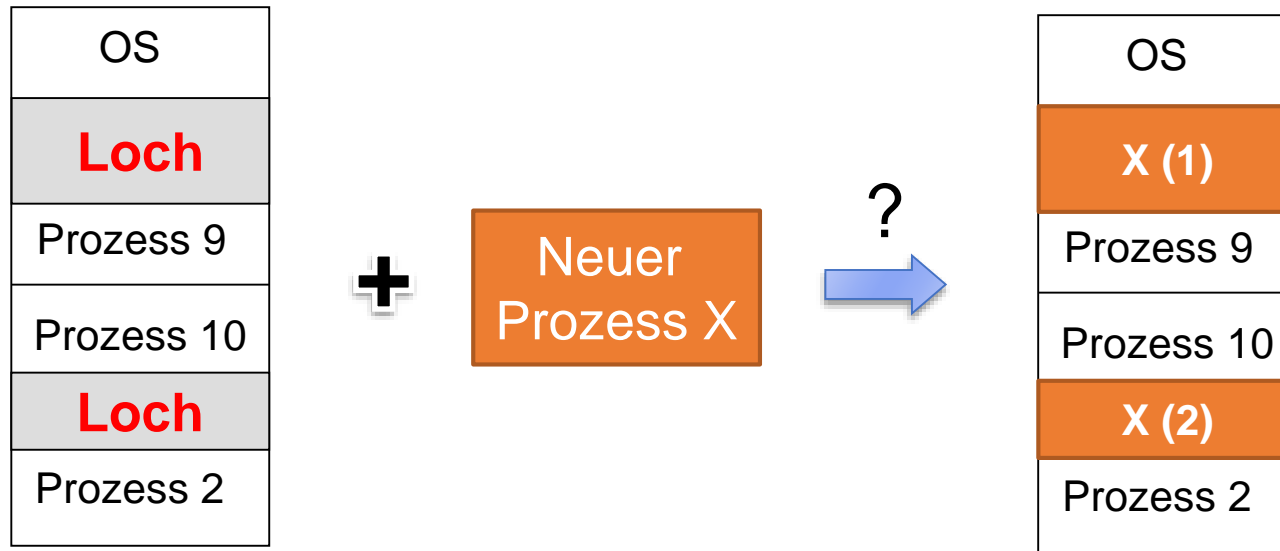
- ▶ Es bleibt dennoch Problem P2:
 - ▶ Programmcode+Daten müssen „hintereinander im Speicher liegen
 - ▶ => Problem des **zusammenhängenden Speichers**

Was ist Korrekt? (Moderne BS)

- ▶ A. Wenn ein Programm terminiert, werden alle Prozesse mit höherem Adressraum „nach unten“ verschoben, um den freien RAM zu vergrößern
- ▶ B. Durch eine MMU mit Basis- und Limitregister können wir verhindern, dass ein Prozess die Daten anderer Prozesse sieht oder manipuliert
- ▶ C. Die Speicherplatzverschwendung bei der internen Fragmentierung beträgt im Schnitt die halbe Größe eines Speicherslots (d.h. atomaren „Speicherblocks“)
- ▶ D. Die Adressenübersetzung beeinträchtigt nicht die Leistung eines Computers
- ▶ E. Helge Schneider mag ein toller Musiker sein, aber von externer Fragmentierung hat er keine Ahnung

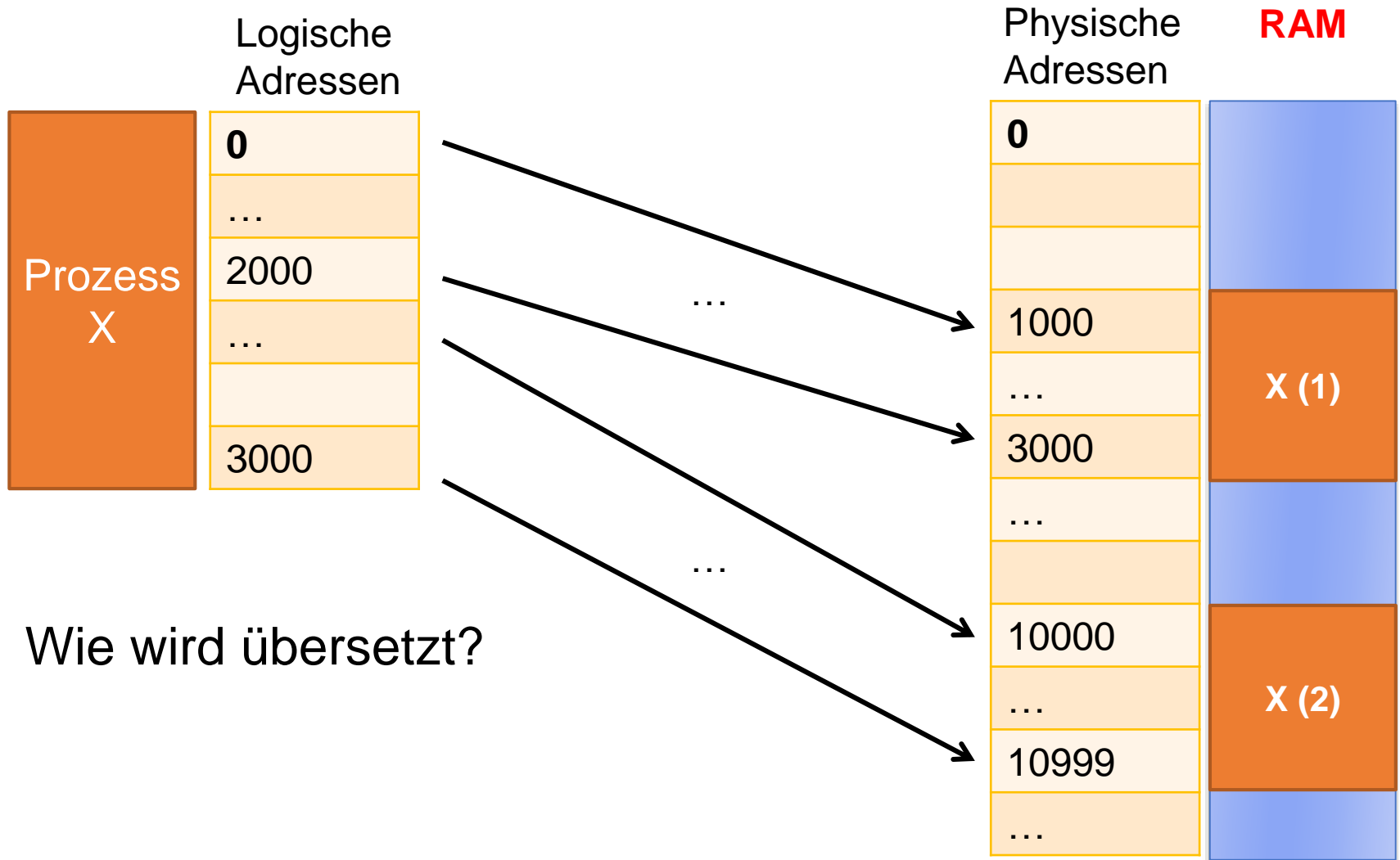
Paging - Grundkonzepte

Naive Lösung

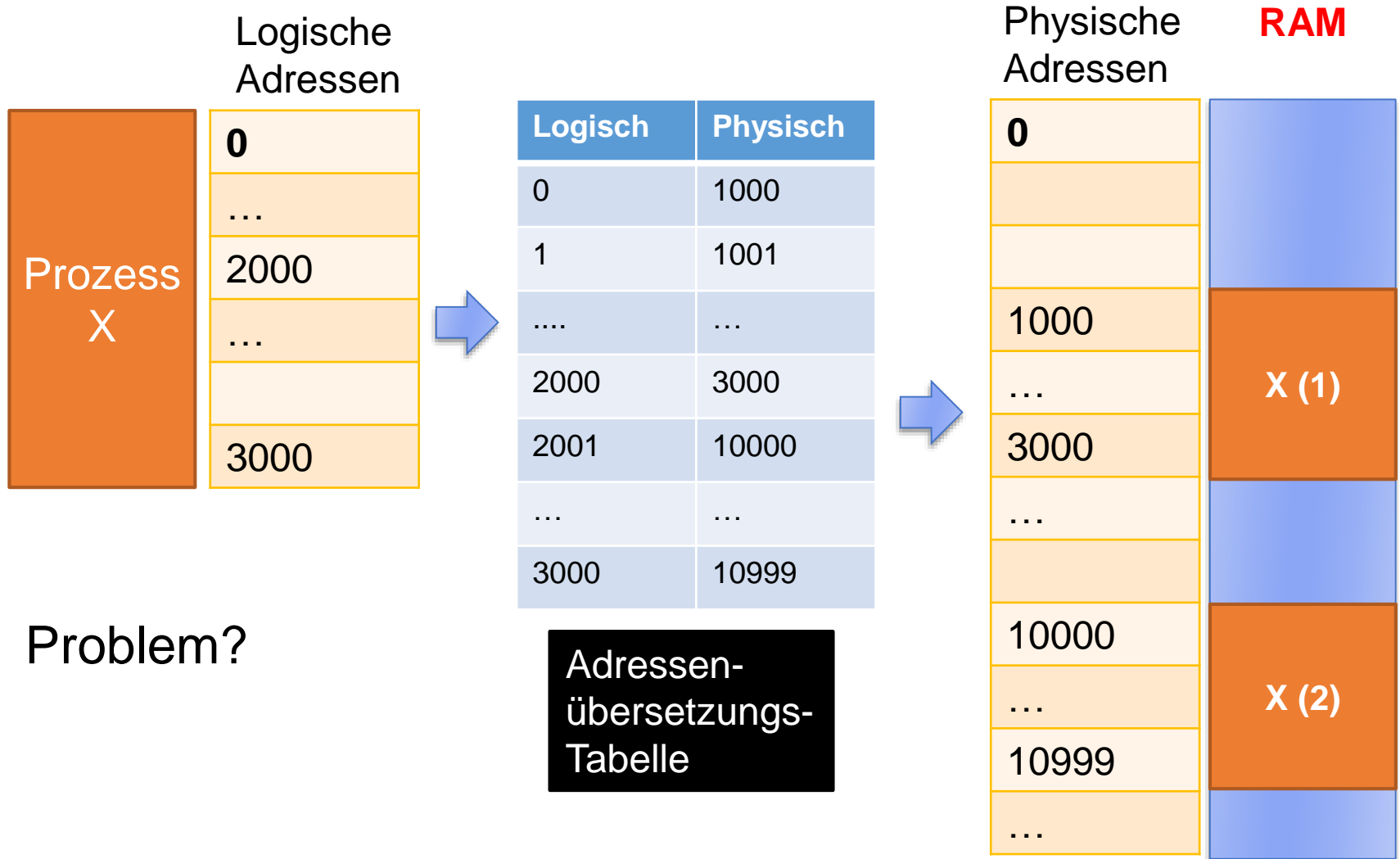


- ▶ Wie können wir erreichen, dass Prozess X einen zusammenhängenden logischen Adressraum hat?
- ▶ Wir übersetzen jede einzelne logische Adresse von X in eine (frei wählbare) physische Adresse

Naive Lösung /2



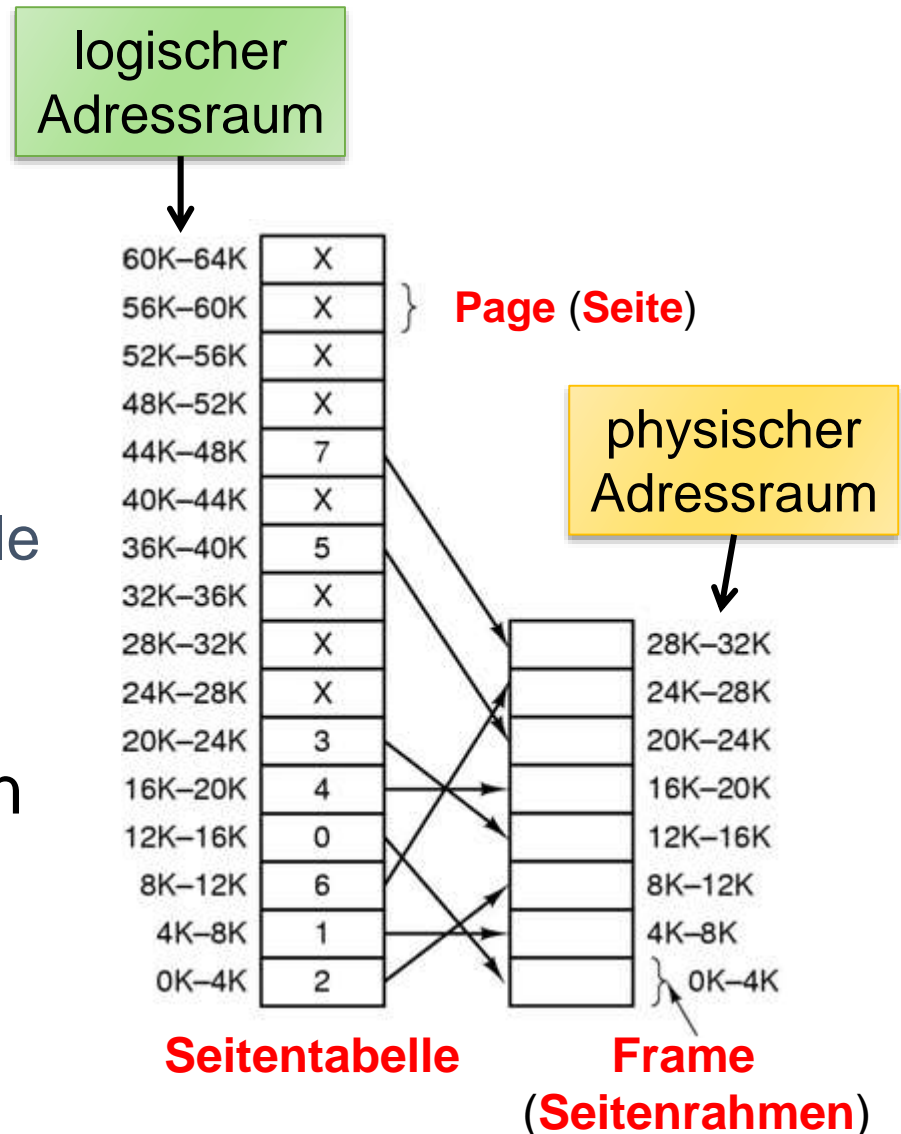
Naive Lösung /2



Problem?

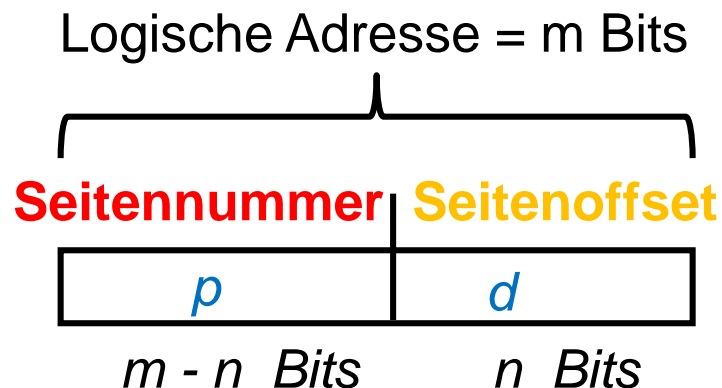
Paging („Kachelverwaltung“)

- ▶ Definition von **Paging**:
 - ▶ Methode der Arbeits-speicherverwaltung durch Übersetzung der Adressen zur Laufzeit
 - ▶ Dies geschieht mit einer Adress-Übersetzungstabelle (**Seitentabelle**)
- ▶ Blöcke der logischen Adressen (**Pages**) werden in Blöcke der physischen Adressen (**Frames**) übersetzt



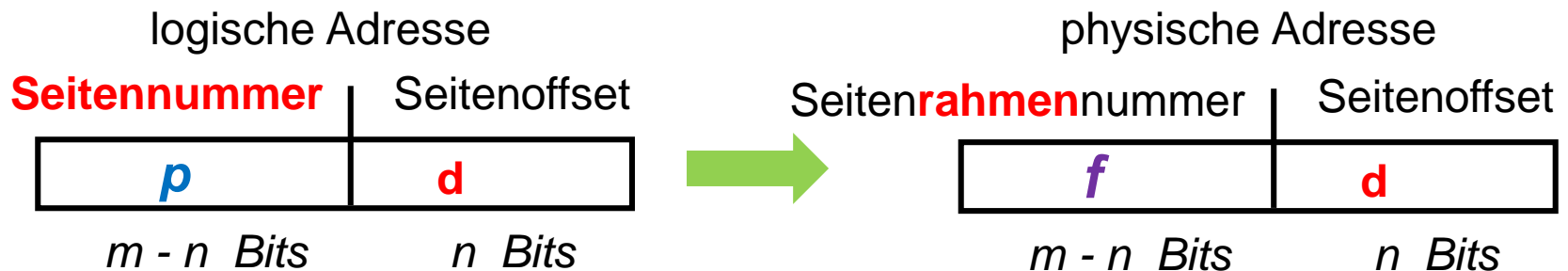
Addressübersetzung /1

- ▶ Der Kern des Pagings ist die Adressübersetzung
- ▶ Dazu wird eine von der CPU erzeugte Adresse (logische Adresse) „konzeptionell“ unterteilt in
 - ▶ **Seitennummer** bzw. **Seitenindex** (**page number**) **p**
 - ▶ Obere Bits der logischen Adresse
 - ▶ **Seitenoffset** (**page offset**) **d**
 - ▶ Untere Bits der log. Adresse)
- ▶ Bei Adressraum der Größe 2^m und 2^n als Größe einer Seite haben wir:



Addressübersetzung /2

- ▶ Die Addressübersetzung funktioniert durch das Ersetzen (in der Adresse) der Seitennummer **p** durch eine (eigentlich beliebige) Seiten**rahmen**nummer **f** (d.h. obere Bits einer physischen Adresse)
- ▶ Die n Bits des Seitenoffsets **d** werden direkt übernommen



- ▶ Hauptproblem: wie kann man das schnell machen?
 - ▶ Speicherzugriffe müssen sehr schnell erfolgen

Adressübersetzung durch Seitentabelle

Logischer Adressraum:
Adressen, die das Programm „sieht“

Index =
Seiten-
nummer **p**

Inhalt =
Seiten-
rahmen-
nummer **f**

Index des Seitenrahmens =
obere Bits der
physischen Adresse

0	Daten Seite 0
1	Daten Seite 1
2	Daten Seite 2
3	Daten Seite 3

Eintrag 0

Eintrag 1

Eintrag 3

1
4
3
7

Seitentabelle

Seitenindex („Kachel-nr.“) **p**

0	
1	Inhalt Seite 0
2	
3	Inhalt Seite 2
4	Inhalt Seite 1
5	
6	
7	Inhalt Seite 3

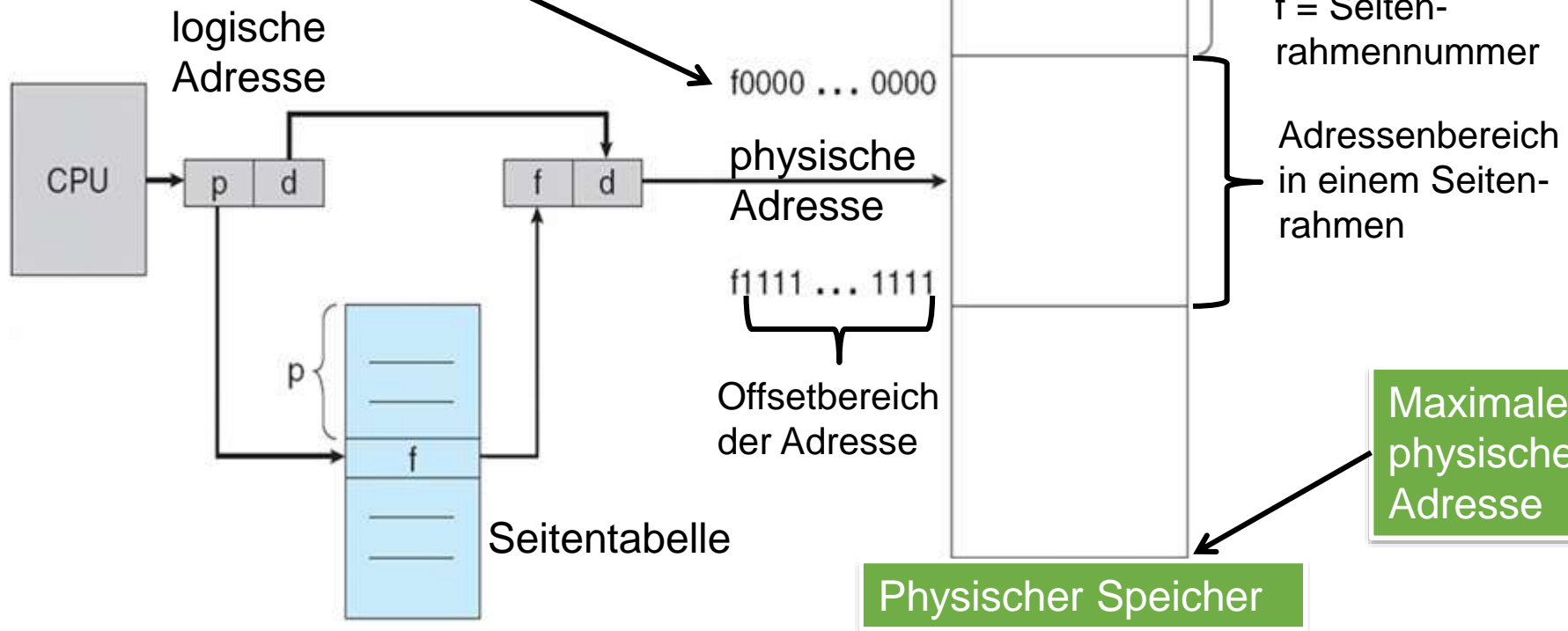
Physischer Adressraum:
die wirklichen
(Hardware)
Adressen

Physischer Speicher

- ▶ Die Seitennummer **p** wird als **Index** des Eintrags in der Seitentabelle benutzt
- ▶ Der Inhalt des Eintrags ersetzt die oberen Bits der logischen Adresse

Adressübersetzung durch Seitentabelle /2

f = Seitenrahmennummer = die oberen Bits der Basisadresse eines Seitenrahmens



- ▶ Größe einer Seite == Größe eines Seitenrahmens
- ▶ IA-32 Architektur (x86): 4 kB oder 4 MB

Video: Page Tables

- ▶ Virtual Memory: 5 Page Tables [09a]
 - ▶ <https://www.youtube.com/watch?v=KNUJhZCQZ9c>
 - ▶ Ab 4:48 bis Ende (Min:sec)
 - ▶ Sehr empfehlenswert: ganzes Video ansehen!

Coarse-grained: pages instead of words

31

- The **Page Table** manages larger chunks (**pages**) of data:
 - Fewer **Page Table Entries** needed to cover the whole address space
 - But, less flexibility in how to use the RAM (have to move a page at a time)
- Today:
 - Typically 4kB pages (1024 words per page)
 - Sometimes 2MB pages (524,288 words per page)

Q: How many entries do we need in our Page Table with 4kB pages on a 32-bit machine?

- 1 for every word = $2^{30} = 1$ billion
- 1 for every 1024 words = 1 million
- 1 for every 4096 words = 262,144

A: 1 for every 1024 words = 1 million
With 4kB pages we have 1024 words per page. That means we need 1 Page Table Entry for every 1024 words. In total we have 1 billion words, so $1 \text{ billion} \div 1024$ is 1 million Page Table Entries. This is much more manageable.

Coarse-grain:
maps chunks
of address

Page Table
translates VA → PA

Map	
VA	PA
0-4095	4096-8191



Zusammenfassung Adressübersetzung

- ▶ Eine von der CPU erzeugte Adresse (logische Adresse) wird unterteilt in
 - ▶ **Seitennummer** bzw. **Seitenindex** (**page number**) **p**
 - ▶ Verwendet als Index eines Eintrags der Seitentabelle
 - ▶ Die Seitentabelle enthält **Seitenrahmennummern** **f** (**frame numbers**), d.h. obere Bits von Basisadressen der Seitenrahmen
 - ▶ **Seitenoffset** (**page offset**) **d**
 - ▶ Offset der Adresse innerhalb einer Seite
 - ▶ D.h. „Abstand“ der Adresse zur nächstkleineren Basisadresse
- ▶ Bei der Adressübersetzung wird der Eintrag der Seitentabelle mit Index **p** nachgeschaut, und die oberen Bits der Adresse durch seinen Inhalt **f** ersetzt



Paging und Fragmentierung

- ▶ Wie hilft Paging, die Fragmentierung zu beseitigen? (externe oder interne?)
- ▶ Um ein Prozess der Größe „n Seiten“ auszuführen:
 - ▶ Finde n freie Seitenrahmen
 - ▶ Weise sie dem Prozess via Seitentabelle als Speicher zu
- ▶ Beseitigung der externen Fragmentierung
 - ▶ Der Clou: die Frames müssen nicht zusammenhängend sein, sondern können ggf. in vielen verschiedenen Bereichen des physischen Speichers liegen!
 - ▶ Da jeder Seitenrahmen gleich groß ist, gibt es keine Speicherlücken mehr
- ▶ Welches Problem bleibt noch?

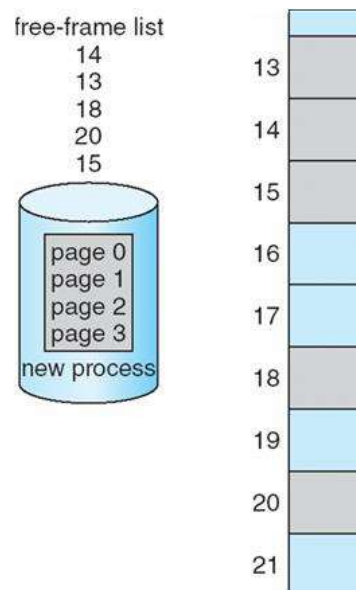
Hat jeder Prozess eine eigene Seitentabelle?

- ▶ Verschiedene Prozesse benutzen i.A. denselben logischen Adressraum (der oft bei 0 anfängt) mit versch. Daten. Zugleich wird der Eintrag in der Seitentabelle durch (obere) Bits einer logischen Adresse ausgewählt. Gäbe es nur eine Seitentabelle, würden verschiedene Prozesse ggf. den gleichen Eintrag bei Adressenübersetzung der Seitentabelle holen (=> zur gleichen physischen Adresse übersetzen), was nicht gewünscht ist.
- ▶ Die Seitentabelle könnte aber so implementiert sein, dass man bei jedem Eintrag zwischen verschiedenen Prozessen unterscheidet, d.h. ein "Prozess-ID-Label" den Index der Seitentabelle erweitert. Bei den gängigen Prozessoren wie die x86-Architektur ist das aber nicht der Fall, das zeigt ein Blick auf die MMU von x86 (z.B. [hier](#) - auch eine interessante Lektüre an sich).
- ▶ Es wird bei Linux tatsächlich der Inhalt des CR3 Systemregisters bei einem Taskwechsel ggf. ausgetauscht (siehe [hier](#) unter 6.3, "change Memory context (change CR3 value)"). Dieser Register enthält die Basisadresse der Seitentabelle (d.h. PTBR, siehe [hier](#)).
- ▶ Es ist aber denkbar, dass andere Architekturen / BS mit einer einzigen Seitentabelle und Labels pro Prozess arbeiten (das wird z.T. auch bei *invertierten* Seitentabellen gemacht) (siehe [hier](#)).
- ▶ Wichtig ist nur, dass es für jeden Prozess eine separate *Abbildung* von logischen auf die physische Adressen möglich ist. **Wir können also vereinfachend sagen, dass es eine Seitentabelle pro Prozess gibt.**

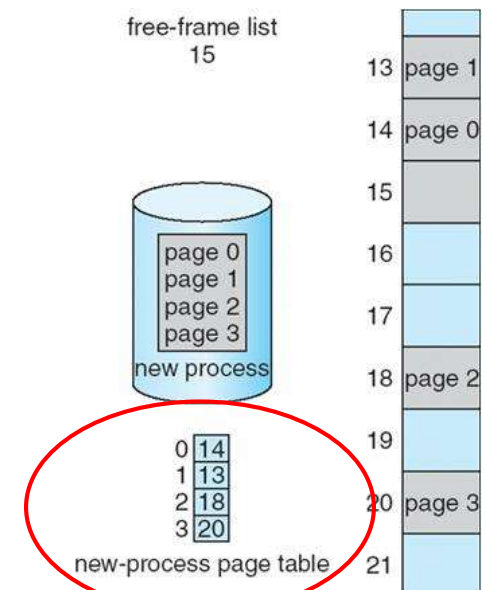
Das ist Antwort auf eine Frage aus dem
WS 2011/12 – hier als Ergänzung

Paging und Prozesse

- ▶ Mit jedem Prozess wird eine neue Seitentabelle nur für diesen Prozess erzeugt
 - ▶ Die nötigen Seitenrahmen können überall liegen
- ▶ Aber das BS muss die Verwendung des physischen Speichers nachverfolgen
- ▶ Dazu dient die (globale) **frame table** (**Seitenrahmen-tabelle**) mit Informationen wie ...
- ▶ Welche Seitenrahmen sind belegt und welche frei?
- ▶ Falls belegt, zu welchem Prozess / welchen Prozessen gehört der Rahmen?



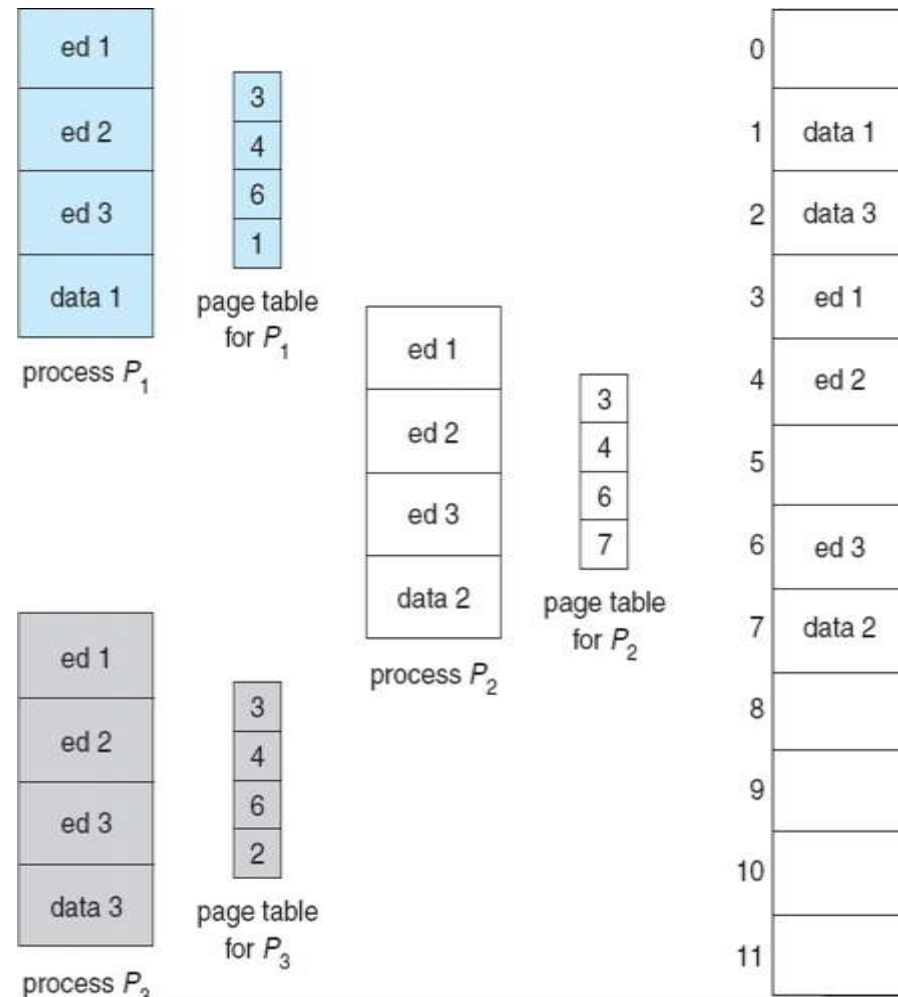
vor dem Laden



nach dem Laden

Gemeinsam Benutzte Seiten (Shared Pages)

- ▶ Nur-Lese Code (**reentrant Code**) kann gemeinsam von mehreren Prozessen genutzt werden
 - ▶ Z.B. DLL's
- ▶ Auch gemeinsame Daten:
 - ▶ Shared-Memory Bereiche für IPC
- ▶ Auch hier wird Paging benutzt:
 - ▶ Mehrere Prozesse haben Seitentabellen mit gleichen Seitenrahmennummern (hier ed 1 -- ed 3)



Zusammenfassung

- ▶ Management des Speichers
 - ▶ Zwei Herausforderungen
 - ▶ Probleme mit den Speicherlöchern
 - ▶ Relocation des Code
- ▶ Paging – Grundkonzepte
 - ▶ Adressübersetzung, Seitentabellen
- ▶ Quellen
 - ▶ Speicher: Silberschatz et al., Kap. 8+9; Tanenbaum Kap. 3.2 + 3.3

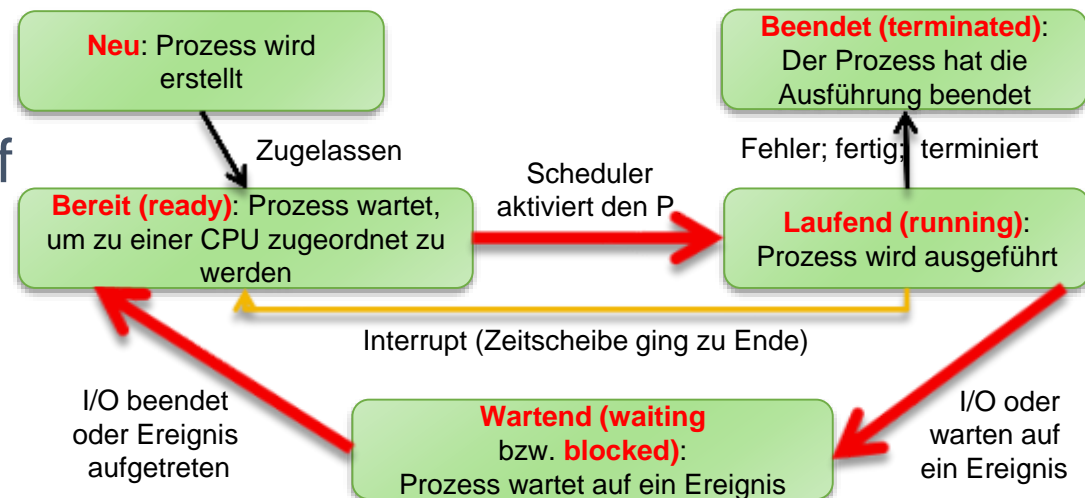
Zusätzliche Folien

Swapping und Prozessverwaltung

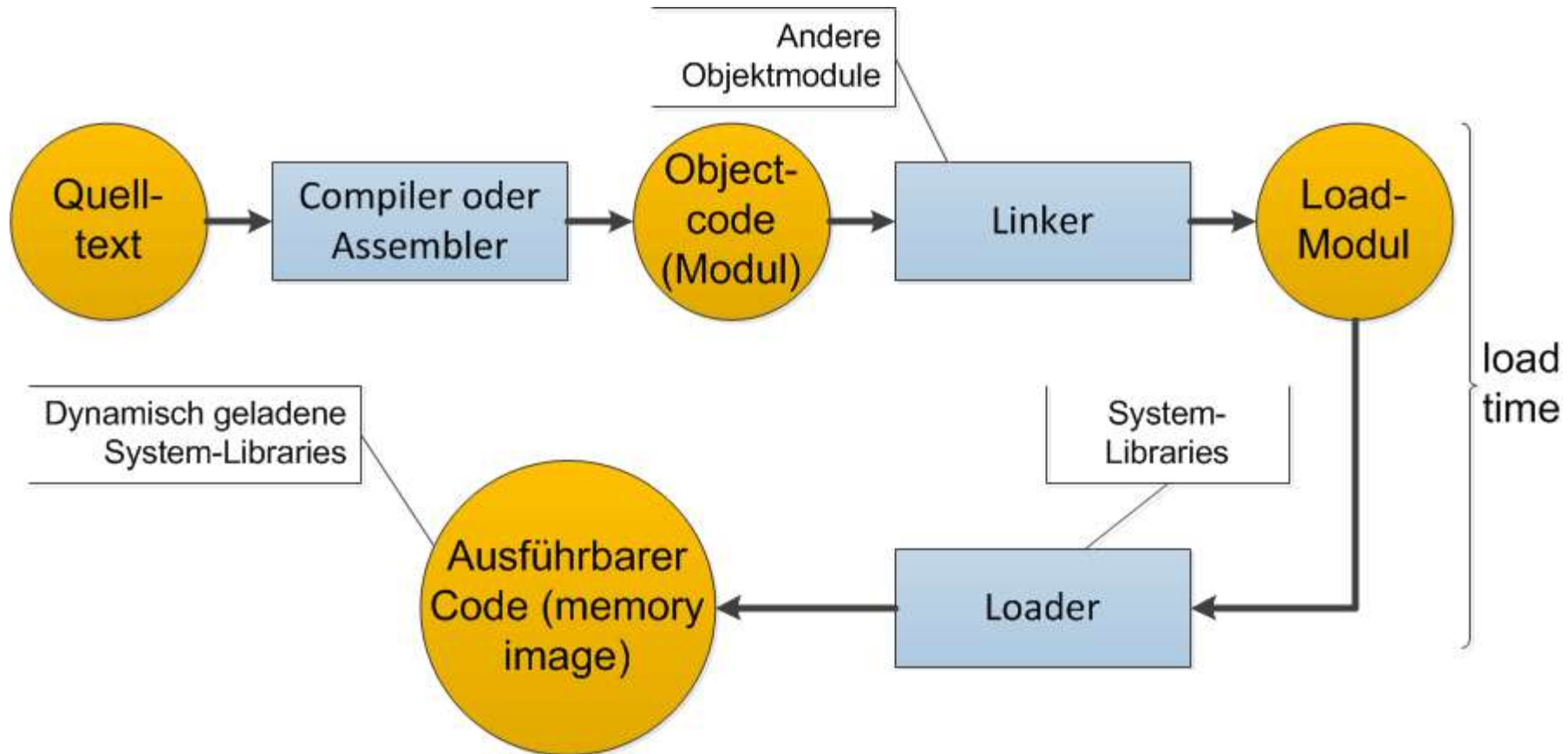
- ▶ Man lagert zunächst solche Prozesse aus, die auf ein Ereignis warten (blocked)
 - ▶ Wenn ein ausgelagerter Prozess in den Hauptspeicher zurückgebracht wird, sollte dieser nicht mehr blockiert sein
- ▶ Also reichen die 3 (Haupt-)Prozesszustände **ready**, **running**, **waiting (blocked)** nicht mehr aus!
- ▶ Wir müssen unterscheiden:

- ▶ **waiting / blocked**: Prozess kann nicht ausführen, weil er auf Ereignis (I/O, ...) wartet

- ▶ **suspended**: P. kann nicht ausführen, weil er ausgelagert ist



Kompilieren und Laden eines Programms



Möglichkeiten der Adressanpassung

- ▶ **Zur Compiler- / Assemblierzeit:** Lage des Codes im Speicher muss a priori bekannt sein
- ▶ **Zur Ladezeit** (z.B. Atari ST)
 - ▶ Das Programm **Loader** geht direkt nach dem Laden durch den Code und passt es an die Zieladresse an
 - ▶ Compiler bzw. Linker muss das unterstützen
- ▶ **Zur Laufzeit**, durch sog. **relativen Code**
 - ▶ D.h. statt JUMP 48335, verwende JUMP (PC)+631
- ▶ **Zur Laufzeit**, für jeden Code – durch Hardware
 - ▶ Das ist die flexibelste Lösung => Memory Management Unit