



**Hochschule
Augsburg** University of
Applied Sciences

Vorlesung: Betriebssysteme

Speicherverwaltung

Prof. Dr. Lothar Braun, Dr.-Ing. Volodymyr Brovkov
Sommersemester 2024

Einführung

Verwaltung von freiem Speicher

Virtueller Speicher: Grundlagen

Virtueller Speicher: Ein- und Auslagerung

Laden und Ausführen von Programmen

Einführung

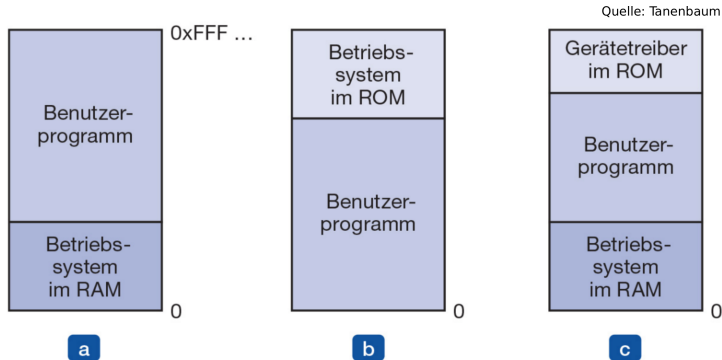
Verwaltung von freiem Speicher

Virtueller Speicher: Grundlagen

Virtueller Speicher: Ein- und Auslagerung

Laden und Ausführen von Programmen

- **Früher: Betriebssysteme erlauben direkten Zugriff auf den Speicher**

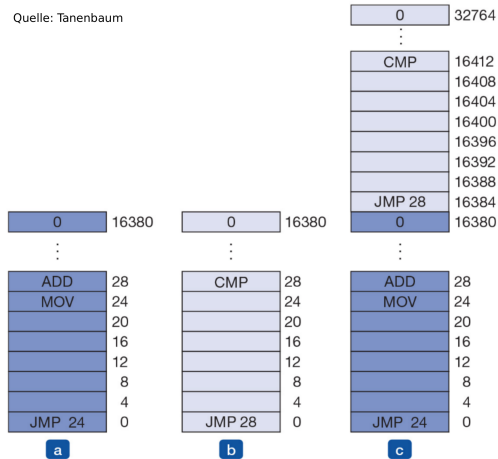


Verschiedene Konzepte für direkten Speicherzugriff

Ausführen mehrerer Programme bei direktem Speicherzugriff

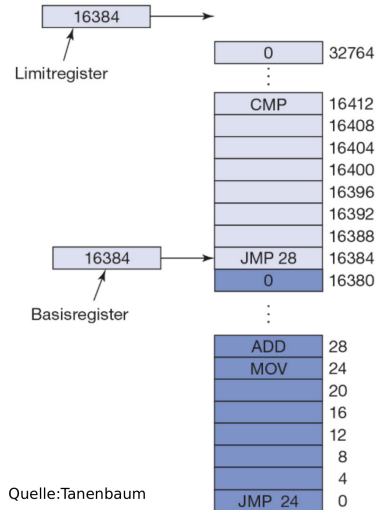


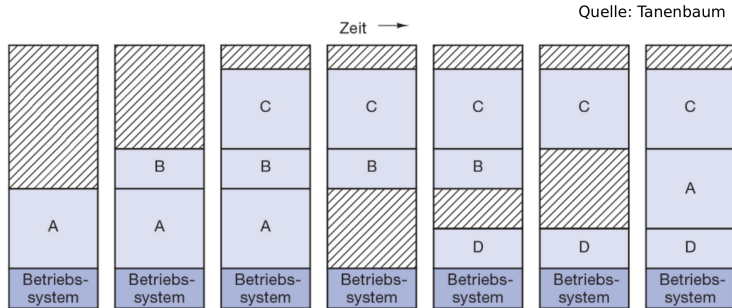
Quelle: Tanenbaum



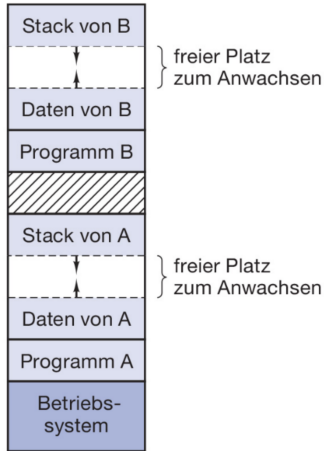
- **Ausführen mehrere Programme (Multiprogrammierung) schwierig**
- **Programme können auf fremden Speicher zugreifen**
 - Kann Betriebssystem oder andere Programme zum Absturz bringen
- **Lösung: Eigene Adressräume für jeden Prozess**

- **Prozesse werden an Offsets geladen**
 - Basis- und Limitregister enthalten Offset und neues Limit
- **Code adressiert Speicheradressen direkt**
- **Hardware übersetzt Zugriff auf Adressen mit Hilfe von Basis- und Limitregistern**
- **Jeder Prozess hat eigenen *Adressraum* innerhalb des Speichers**





- **Problem: Ausführen mehrerer Prozesse kann mehr Speicher benötigen als vorhanden**
 - Lösung: Auslagern von Prozessen wenn andere Prozesse lauffähig werden



- **Allokation von Speicher zur Laufzeit**

- Heap wächst/schrumpft bei Aufrufen von `malloc()/free()`
- Stack wächst/schrumpft bei Funktionsaufrufen

- **Freier Speicher muss verwaltet werden**

- Platz muss bei Allokation gesucht werden
- Freigegebener Speicher muss wiederverwendet werden können

Einführung

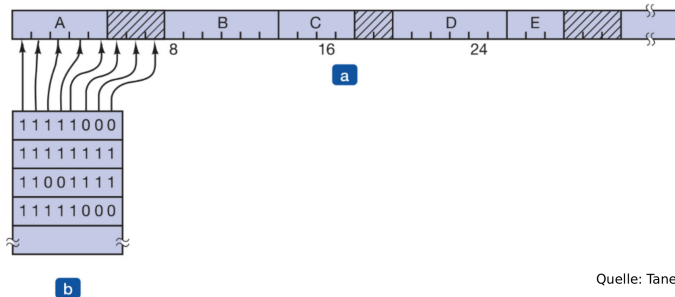
Verwaltung von freiem Speicher

Virtueller Speicher: Grundlagen

Virtueller Speicher: Ein- und Auslagerung

Laden und Ausführen von Programmen

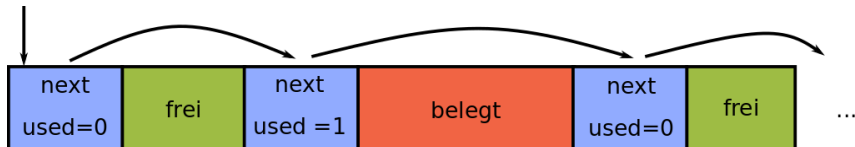
- **Zuteilung eines Speicherabschnitts fester vorgegebener Größe für Zeitraum**
- **Beispiel:**
 - malloc() / free() in der Standard-C-Bibliothek
 - New/delete-Operatoren in C++ und Java
 - Zuteilung des physikalischen und Auslagerungs-Speichers im Betriebssystem
 - Zuteilung des Platzes auf einer Festplatte
- **Typischer Fall: Zuteilung von Speicher variabler Größe**



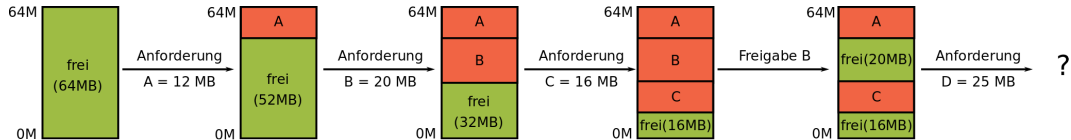
Quelle: Tanenbaum

- **Aufteilung des Speichers in Blöcke**

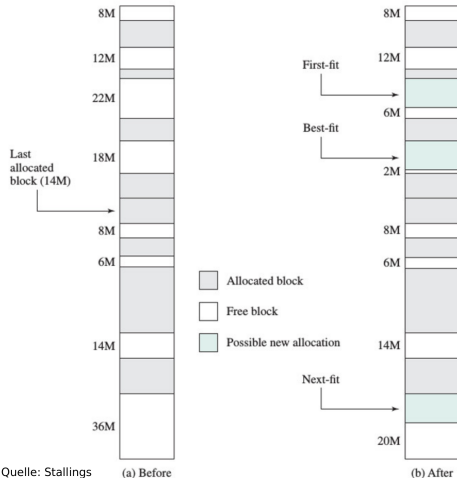
- Jeder Block kann belegt oder frei sein
- Belegungsinformation wird in Bitmap abgelegt → Durchsuchen des Bitmaps bei Allokation



- **Speicherbereiche haben beliebige Länge**
- **Suche nach freiem Speicher durch Suche in Liste**
 - Wenn freier Platz in passender Größe gefunden, dann als belegt markieren
 - Aufteilen falls freies Segment deutlich größer als Bedarf
- **Freigabe:**
 - Markieren als nicht benutzt
 - Ggf. zusammenfassen mit freien Speicherbereichen vorher und nachher



- **Verschiedene Strategien existieren um Fragmentierung gering zu halten**



- **First Fit: Erster passender Speicher wird gewählt**

- Suchaufwand steigt mit vielen allokierten Bereichen
- Fragmentierung am Anfang, großer freier Block am Ende

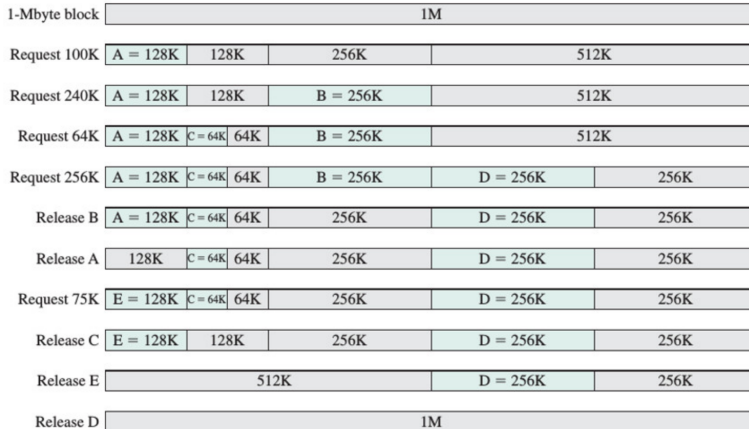
- **Best fit: Kleinster passender Block**

- Langsamer als First Fit
- Produziert viele kleine nicht weiter verwendbare Lücken

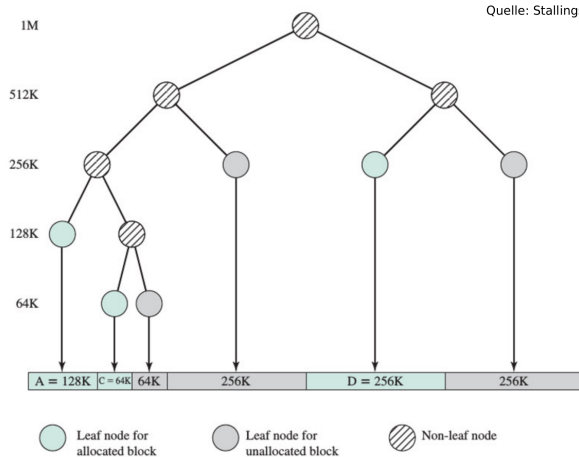
- **Next fit: First Fit nach vorheriger Allokation**

- Verkleinert großen Block am Ende

- Optimierung: Organisation des Speichers in Blöcken



Quelle: Stallings



- Listen für jede verfügbare Blockgröße können vorgehalten werden

- **Speicher wird in Blöcke der Länge 2^n aufgeteilt**
 - Initial: Ein großer Block des gesamten Speichers
 - Bei Anforderung von m Bytes:
 - Suche nach kleinstem freiem Block mit Größe von m oder mehr Bytes
 - Halbierung des Blocks falls halbiertes Block $\geq m$
 - Wiederholung des vorherigen Schrittes bis keine Halbierung mehr möglich ist
 - Freigabe von Speicher:
 - Freigabe des Blocks und ggf. Zusammenfügen zweier nebeneinander liegender Blöcke gleicher Größe
- **Schnell: Anforderung und Freigabe in $O(N)$**
- **Potentiell hoher Speicherverschnitt von Speicher aufgrund *interner Fragmentierung***
 - Problem: Nur Blöcke der Größe 2^n können allokiert werden
- **Verwendung z.B. in Betriebssystem zur Verwaltung des physikalischen Speichers**
 - Fragmentierung wegen *virtueller Speicherverwaltung* kein Problem

Einführung

Verwaltung von freiem Speicher

Virtueller Speicher: Grundlagen

Virtueller Speicher: Ein- und Auslagerung

Laden und Ausführen von Programmen

- **Relokation**

- Programme werden an beliebiger Stelle im (physikalischen) Speicher abgelegt
- Addressumsetzung bei Ausführung

- **Teilen von Daten**

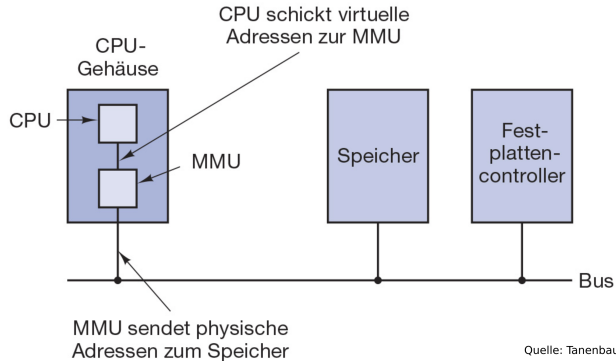
- Logische Ebene: Gemeinsamer Speicherbereich (Shared Memory)
- Physikalische Ebene: Gleicher physikalischer Speicher für gleiche Daten

- **Schutz**

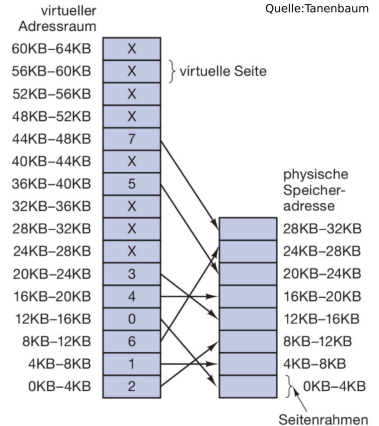
- Prozesse dürfen nicht auf Daten anderer Prozesse zugreifen
- Hardware-Unterstützung nötig

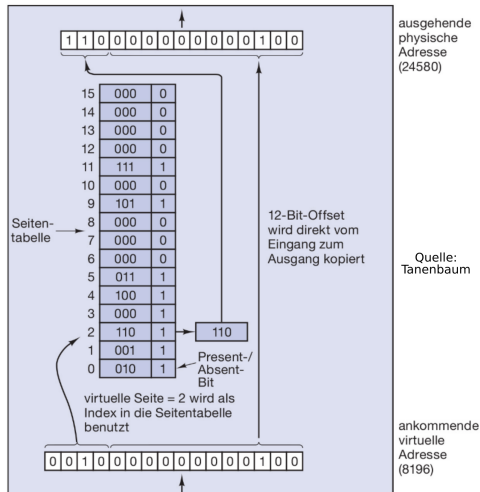
- **Logische Organisation**
 - Bereitstellen eines (virtuellen) Adressraumes mit Segmenten
 - Variable Größe der Segmente für Daten, Stack, ...
 - Zugriffsberechtigung für einzelne Sektionen ermöglichen (z.B. Code-Ausführung nur in Text-Segmenten)
- **Organisation des physikalischen Speichers**
 - Verwaltung von Speicher
 - physikalischer Speicher (RAM)
 - sekundärer Speicher (Auslagerungs-Datei/Partition, ...)
 - Transparentes Management für Prozesse

- **Prozess bekommt einen exklusiven virtuellen Adressraum**
 - Alle vom Programm verwendeten Adressen sind *logische* Adressen
 - Hardware übersetzt *logische* Adressen in *physikalische* Adressen

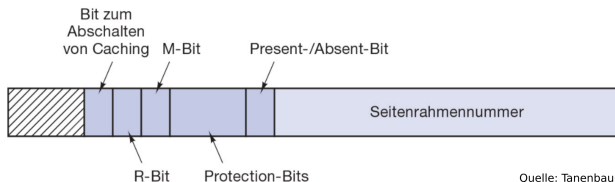


- **Virtueller Speicher wird unterteilt**
 - Aufteilung in Seiten (oder *Pages*) → *Paging*
 - Virtuelle Seite hat physikalische Seiten
 - Physikalische Seiten nicht aufeinander folgend
- **Begriffe**
 - **Seite:** Datenblock konstanter Größe (z.B. 4KB)
 - **Rahmen:** physikalischer Speicherbereich
 - **virtuelle Seite:** logischer Adressbereich
- **Memory Management Unit (MMU)**
 - Umsetzen: virtuelle Adresse → physikalische Adresse
 - Auslösung von Exceptions im Fehlerfall



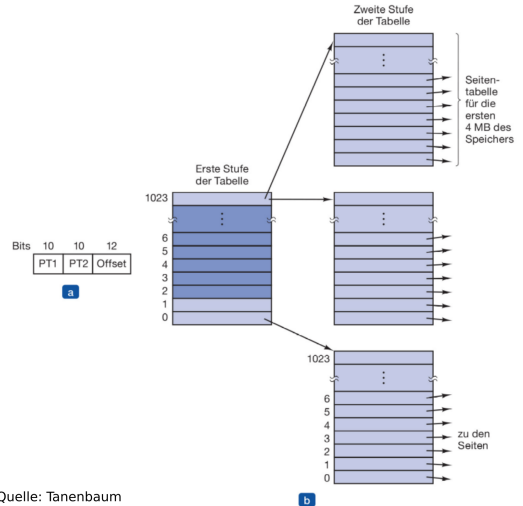


- **Aufteilung der virtuellen Adresse**
 - Erster Teil: Zeiger in *Seitentabelle*
 - Zweier Teil: Offset in Rahmen
- **Seitentabelle muss durch Anforderungen befüllt werden**



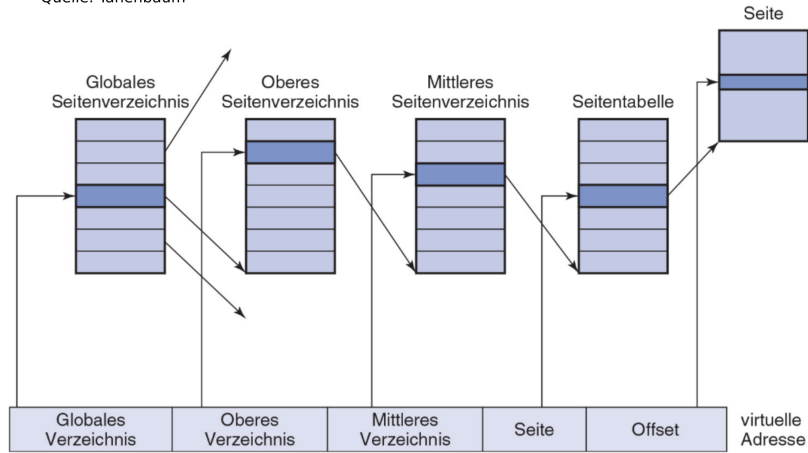
- **Seitenrahmennummer**
 - Nummer des Seitenrahmens (→ physikalische Adresse)
 - Wenn *Present*-Bit gesetzt, dann Zugriff möglich, sonst Seitenfehler
- **Protection-Bits**
 - Zugriffsrechte: *lesen, schreiben, ausführen*
- **M-Bit**
 - Hardware setzt Bit bei Schreibzugriff
- **R-Bit**
 - wird von Hardware bei jedem Zugriff gesetzt → Verwendung bei Auslagerung

- **Tabelle bei viel Speicher sehr groß**
- **Programme verwenden typischerweise kleine Speicherbereiche auf einmal**
- **Mehrstufige Tabellen: Nur Teile der Tabelle müssen gleichzeitig im Speicher gehalten werden**



Quelle: Tanenbaum

Quelle: Tanenbaum



- **Problem: Laufzeit der Adressumsetzung**
 - Bei Speicher-Zugriff müssen mehrere Ebenen an Seitentabellen durchlaufen werden
- **Lösung: Cache in Hardware für oft benutzte Seiten**

Gültig	Virtuelle Seite	Verändert	Schutz	Seitenrahmen
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Einführung

Verwaltung von freiem Speicher

Virtueller Speicher: Grundlagen

Virtueller Speicher: Ein- und Auslagerung

Laden und Ausführen von Programmen

- **Jeder Prozess hat großen Adressraum zur Verfügung**
 - Einzelner Adressraum kann größer sein als physikalisch vorhandener Speicher
 - Physikalische Seiten werden nur nach Anforderung allokiert?
 - Nicht alle Seiten werden zu jedem Zeitraum gebraucht
- **Prozesse können insgesamt mehr RAM anfragen als vorhanden**
 - Benötigte Seiten werden im **primären Speicher** (z.B. RAM) vorgehalten
 - Inaktive Seiten werden in **sekundären Speicher** gehalten
 - Daten in Swap-Partition
 - Code aus ausführbaren Dateien
- **Strategie für Ein-/Auslagerung notwendig**
 - *Einlagern*: Seite im sekundären Speicher wird bei Bedarf in primären Speicher geladen
 - *Auslagern*: Nicht mehr benötigte Seite wird in sekundären Speicher geschrieben

- **Im physikalischen Speicher**

- Präsent
 - unverändert
 - verändert (M-Bit gesetzt)
 - gesperrt (darf nicht ausgelagert werden)
- nicht präsent aber gültig
- ungültig (= dem Prozess vom Betriebssystem nicht zugeteilt)

- **Im sekundären Speicher**

- Nicht vorhanden (= es existiert keine Kopie auf der Platte)
- In Auslagerungsdatei
- In Datei (= es existiert eine benannte Datei mit Daten)

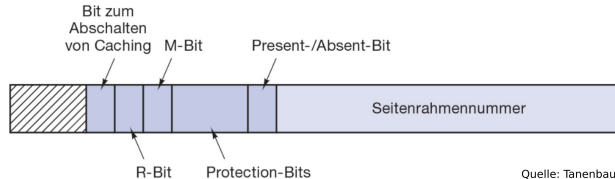
- **Grund: Prozess referenziert Seite die nicht im physikalischen Speicher liegt**
 - Information in Seiteneintrag kodiert → Seitenfehler
 - Betriebssystem prüft ob Seite gültig → Einlagern wenn gültig
- **Einlagern bei Bedarf (*Demand Paging*)**
 - Physikalischer Speicher wird nie vor Benutzung belegt (auch nicht bei malloc(), usw.)
 - *Page Fault Handler* allokiert oder liest von sekundärem Speicher nur die angefragte Seite
 - **wird in den meisten gängigen Systemen eingesetzt**
- **Im Voraus einlagern (*Prepaging*)**
 - Lesen mehrerer aufeinander folgender Seiten von Platte evtl. effizienter

- **Eine Datei wird auf virtuelle Adresse abgebildet**
 - Automatisch durch Loader: Laden von Text-Segment beim Start von Programmen
 - Manuell durch Systemaufruf *mmap*
- **Umsetzung**
 - Seiten werden als *nicht präsent aber gültig* angelegt
 - Speicherort ist in benannter Datei
 - Bei Zugriff auf Speicher-Adresse wird Seitenfehler erzeugt
 - *Page Fault Handler* liest Seite aus Datei
- **Vorteile**
 - Nicht benötigte Teile der Datei werden nicht von Platte gelesen
 - Beim Swapping kein Schreiben auf Platte notwendig wenn nicht modifiziert

- `void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset);`
 - Blendet Datei *fd* ab Offset *offset* mit Länge *length* in den Adressraum eines Prozesses ein
- **Varianten (Parameter *flags* in *mmap*)**
 - **MAP_SHARED**
 - Schreiben in den Adressbereich bewirkt Veränderung der Datei
 - **MAP_PRIVATE**
 - Änderung nur für schreibenden Prozess sichtbar
 - Werden nicht auf Platte geschrieben
 - Beim Schreiben wird Referenz auf Originaldatei der Seite durch Auslagerungsdatei ersetzt

- **Manche Daten existieren (logisch) mehrfach**
 - Code + Daten nach `fork()`
 - dynamische Bibliotheken
 - mehrfach speicher-abgebildete Dateien (Modus “privat”)
- **Mehrere Kopien im physikalischen Speicher vorhalten ist Verschwendung**
 - **Lösung:** Mehrere Seitentabellen-Einträge zeigen auf gleichen physikalischen Rahmen
- **Bei Veränderungen der Daten:**
 - **Copy-on-Write:** Seite wird kopiert sobald Schreibzugriff festgestellt wird (nicht vorher)

- **Wann werden Seiten ausgelagert?**
 - Anforderung von physikalischem Speicher kann nicht erfüllt werden
 - Physikalischer Speicher wird knapp (periodische Untersuchung)
- **Frage:**
 - Welche Seite auslagern?
 - Welche Schritte müssen unternommen werden?



- **Analyse des auszulagernden Seiteneintrages**
 - Prüfen ob Seite ausgelagert werden darf
 - Befindet sich die Seite schon im sekundären Speicher?
 - Wurde Sie seither verändert?
- **Auf Platte schreiben bei Bedarf**

- **Ziel**
 - Auswahl der Seiten die möglichst spät oder nie mehr verwendet wird
- **Strategie**
 - Vergangenes Verhalten \rightarrow zukünftiges Verhalten
 - Lokalitätsprinzip
- **Zeitliche Lokalität:**
 - Zugriff auf Adresse vor kurzem \rightarrow Zugriff in naher Zukunft wahrscheinlich
- **Räumliche Lokalität:**
 - Zugriff auf Adresse an Position $X \rightarrow$ Zugriff auf Speicher nahe X

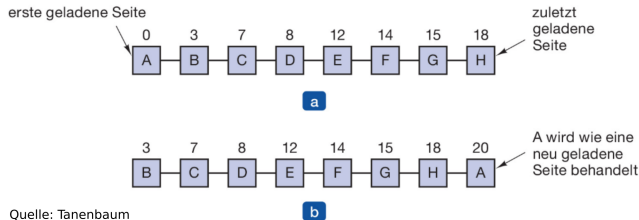
- **Optimale Strategie**
 - Ersetze Seite, die maximale Anzahl von CPU-Instruktionen nicht mehr benötigt wird
 - Voraussetzung: Zukunft muss vorhersagbar sein
- **Least-Recently-Used Algorithmus (LRU)**
 - Ersetze Seite die am längsten nicht verwendet wurde
 - Gute Strategie. Problem: Sehr aufwändig zu realisieren
 - Hardware: Setzen eines Zeitstempel bei jedem Zugriff
 - Software: Durchlaufen der Seitentabelle zur Suche des ältesten Eintrags
- **Not-Recently-Used-Algorithmus (NRU)**
 - Teile Seiten in Klassen anhand des R- und M-Bits in Seiteneintrag ein
 - Klasse 0: nicht referenziert, nicht modifiziert
 - Klasse 1: nicht referenziert, modifiziert
 - Klasse 2: referenziert, nicht modifiziert
 - Klasse 3: referenziert, modifiziert
 - Entferne (zufällige) Seite aus niedrigster Klasse

- **First-In-First-Out Algorithmus (FIFO)**

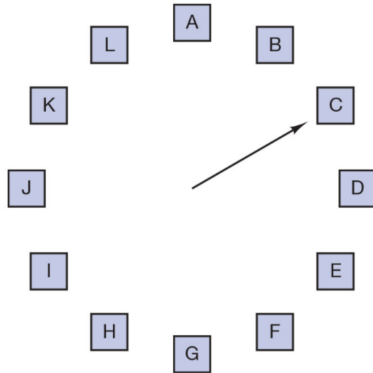
- Erste eingelagerte Seite ist die erste ausgelagerte Seite
- Problem: Beachtet nicht wie häufig eine Seite verwendet wurde → ineffizient

- **Second-Chance Algorithmus**

- Ähnlich wie FIFO-Algorithmus, aber sucht möglichst alte Seite die aber nicht benutzt wird
 - Wenn R-Bit gesetzt, dann R-Bit löschen und Seite an Anfang der FIFO setzen
 - Falls alle Seiten mit R-Bit gesetzt, dann wieder von ältester Seite versuchen



- **Clock-Algorithmus**



Wenn ein Seitenfehler auftritt, wird die Seite untersucht, auf die der Zeiger weist. Die Folgeaktion hängt dann vom R-Bit ab:
R = 0: verdränge die Seite
R = 1: lösche R und rücke Zeiger weiter

Quelle: Tanenbaum

- **Hintergrund:**
 - Veränderte Seiten müssen geschrieben werden → aufwändiger auszulagern
- **Ansatz**
 - Versuche veränderte Seiten so spät wie möglich auszulagern
 - Untersuche *R-Bit* und *M-Bit*
- **Algorithmus**
 - 1) Gehe im Uhrzeigersinn durch Seiten und suche Seiten mit $R\text{-Bit} = M\text{-Bit} = 0$
 - 2) Falls keine Seite in 1) gefunden:
 - Suche im Uhrzeigersinn nach $R\text{-Bit} = 0, M\text{-Bit} = 1$
 - Falls $M\text{-Bit} = 1$ und $R\text{-Bit} = 0$: lagere Seite aus
 - Falls $R\text{-Bit} = 1$: Setze $R\text{-Bit} = 0$
 - 3) Falls in 2) keine Seite gefunden: Alle Seiten haben jetzt $R\text{-Bit} = 0$. Gehe zu 1)

Strategie	Kommentar
Optimal	Nicht realisierbar, aber guter Maßstab
LRU	Sehr guter Algorithmus, aber schwer zu implementieren
NRU	Sehr grobe Annäherung an LRU
FIFO	Einfach zu bauen, entfernt aber oft benötigte Seiten
Second Chance	Deutliche Verbesserung zu FIFO
Uhrzeiger-Algorithmus	Realistische Implementierung
Erweiterter Uhrzeiger-Algorithmus	Guter und effizienter Algorithmus

Einführung

Verwaltung von freiem Speicher

Virtueller Speicher: Grundlagen

Virtueller Speicher: Ein- und Auslagerung

Laden und Ausführen von Programmen

- **Laden eines Programms**

- Allokation von Speicher für einzelne Segmente und schreiben der Programmdateien, oder
- Memory-Mapping von Dateien → Einlagern bei Page Faults

- **Laden von dynamischen Bibliotheken**

- Teilen von Seiten zwischen verschiedenen Prozessen

- **Aufbau des Adressraums eines Prozesses**

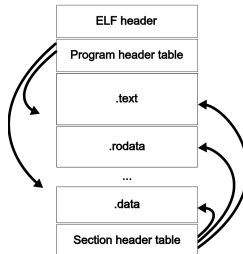
- Code, Heap, Stack, ...

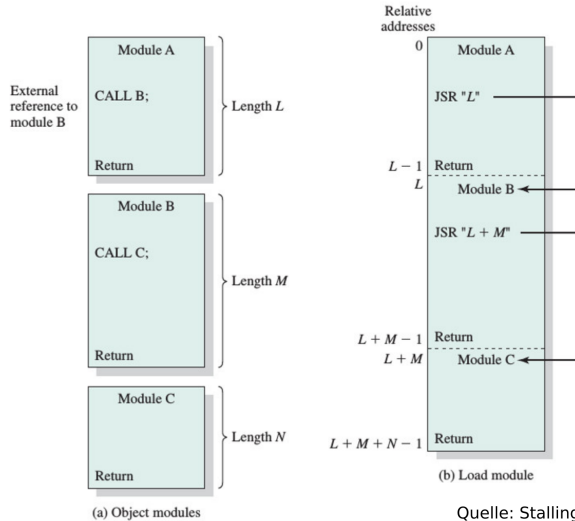
- **Ausführen des Programms:**

- Vergrößerung / Verkleinerung des Stacks und Heaps
- Laden weiterer Bibliotheken (z.B. Plugins) über eigene Systemaufrufe (z.B. `dlopen()`, `dlsym()`, `dlclose()`)

- Programme enthalten sämtliche Informationen, die für den Loader relevant sind
 - Definiertes Format, dass von Betriebssystem-Loader verstanden werden muss
 - Beispiele:
 - Windows: Portable Executables (PE)
 - Linux/Unix: Executable and Linkable Format (ELF)
 - MAC OS X: Mach Object File Format (Mach-O) / Universal File Format

ELF File Format





- **Statisches Binden**

- Linking erfolgt zur Übersetzungszeit
- Linker kopiert den Code der verwendeten Bibliotheken
- Linker passt Adressen an
- Betriebssystem lädt nur ein Modul bei Programmstart → Existenz von Bibliothek unbekannt
- **Nachteil**
 - Vergeudung von Speicherplatz (Festplatte + Hauptspeicher)

- **Dynamisches Binden**

- Binden erfolgt zur Laufzeit: Bei Start oder Ausführung des Programms
- Loader der Betriebssysteme löst Adressen von Symbolen zur Laufzeit aus
- **Vorteil:**
 - Bibliotheken müssen nur einmal vorhanden sein (Festplatte + Hauptspeicher)
- **Nachteil:**
 - Bibliotheken müssen verwaltet werden (Versionierung muss stimmen!)

- **Hauptprogramm: Code + Daten**
- **Dynamische Bibliotheken: jeweils Code + Daten**
- **Heap**
- **Stack**
- **Besondere Bereiche (Shared Memory, Memory Mapped Dateien, ...)**
- **Betriebssystem**

- **Heap**

- Verwaltung des Heaps passiert durch Bibliotheksfunktionen: **malloc**, **free**
- **malloc**, **free** müssen nicht unbedingt für jeden Aufruf Speicher allokieren
- Anpassung (größer / kleiner) der Gesamtgröße des Heap durch Systemaufruf
 - **int brk(void *new_end_of_data_segment);**
 - **void* sbrk(intptr_t increment);**

- **Stack**

- Vergrößerung erfolgt automatisch durch das Betriebssystem
- *Page-Fault Handler* vergrößert Speicher bei Zugriff direkt unterhalb des Stack-Segments

Fragen?