Betriebssysteme I – Sommersemester 2009
Kapitel 6: Speicherverwaltung und Dateisysteme

Hans-Georg Eßer
Hochschule München

Teil 2: Zusammenhängende Speicherzuordnung

06/2009

Partitionen fester, gleicher Größe

- Aufteilen des Hauptspeichers in feste Partitionen gleicher Größe
- Beispiel: 1 GByte RAM, 1024 Partitionen zu je 1 MByte
- Konsequenzen:
  - Maximal 1024 Prozesse im Speicher (abzgl. Platz für BS)
  - Maximale Speichernutzung pro Prozess: 1 MByte
  - bei mehr als 1024 Prozessen: ggf. Swapping

Fragmentierung

Bei Partitionen fester, gleicher Größe:

- Starke **interne Fragmentierung** – ungenutzter Platz innerhalb einer Partition
- keine **externe Fragmentierung** – Speicher vollständig in Partitionen unterteilt

Verwaltung der freien Bereiche

Bei Partitionen fester, gleicher Größe:

- ganz einfach: Partitionen durchnumeriert; zu jeder Partition die Angabe einer Prozess-ID oder -1 (frei)
- Suche nach freier Partition (für neuen Prozess): in dieser übersichtlichen Liste suchen
- Rückgabe einer Partition (Prozessende): Wert in Liste von PID wieder auf 1 setzen

Übertragen auf Platten

- Konzept lässt sich direkt auf Plattenplatz übertragen:
- Festplatte mit 1 GByte, 1024 Partitionen zu je 1 MByte
- Konsequenzen für maximale Dateigröße und Anzahl der Dateien: wie bei Speicher
- Wie sieht dann ein Datenträgerinhaltsverzeichnis aus?

Mehr Flexibilität: verschiedene Größen

- Prozesse haben i.d.R. unterschiedlichen Speicherbedarf („kleine“ und „große“ Prozesse)
- BS sollte Prozessen „passende“ Partitionen anbieten
- z. B. feste Anzahl von Partitionen der Größen 64 KByte, 128 KByte, 256 KByte, . . . , 8 MByte, 16 MByte

Verwaltung freier Bereiche (1/2)

Bei Partitionen fester, verschiedener Größe:

- immer noch leicht: Mehrere Listen für die verschiedenen Partitionsgrößen
- Listeneintrag enthält neben PID/frei auch die Anfangsadresse der Partition
- Beispiel:

  64 KByte: (PID 3, 0), (frei, 64 K), (frei, 128 K), (frei, 192 K)
  128 KByte: (PID 1, 256 K), (frei, 384 K), (PID 2, 512 K), (frei, 640 K)
  256 KByte: (frei, 768 K), (frei, 1024 K), (frei, 1280 K), (frei, 1536 K)
  512 KByte: (frei, 1792 K), . . .
  . . .

  16 MByte: (PID 4, 960 M), (frei, 976 M), (frei, 992 M), (frei, 1008 M)

Verwaltung freier Bereiche (2/2)

- Erzeugen eines neuen Prozesses:
  - `fork` (o. ä.) gibt gewünschte Speichergröße an
  - Suche zunächst in Liste mit kleinsten passenden Partitionen
  - wenn kein Treffer: suche in Liste mit nächst größeren Partitionen
- Rückgabe eines Prozesses:
  - suche PID in allen passenden Listen
  - ändere Listeneintrag auf frei zurück
  - ggf. Verschieben eines Prozesses aus „zu großer“ Partition in eine kleinere

Fragmentierung

Bei Partitionen fester, verschiedener Größe:

- Etwas geringere **interne Fragmentierung** – durch Auswahl einer geeigneten Partition für jeden Prozess wird weniger Platz verschwendet
- keine **externe Fragmentierung** – Speicher vollständig in Partitionen unterteilt

Übertragen auf Platten

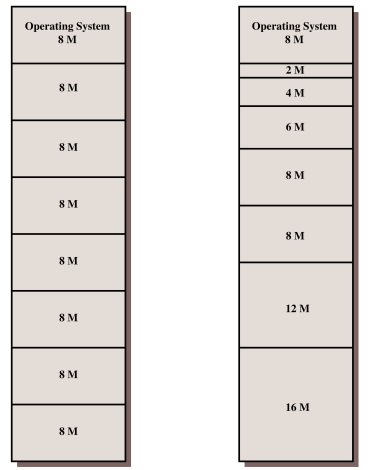
Partitionen fester, verschiedener Größe:

- Auch dieses Konzept direkt auf Dateisysteme übertragbar
- gleiche Konsequenzen: flexiblere Nutzung bei kleinen/großen Dateien
- Datenträgerinhaltsverzeichnis: unverändert (Dateiname, Anfang, Länge)

Vergleich: gleiche/verschiedene Größe

- Gleiche Größe
  - Verschwendung bei kleinen Programmen
  - Programme passen in jede freie Partitition
- Verschiedene Größe
  - Bessere Speicherausnutzung
  - Evtl. ungeschickte Belegung

Bild: Stallings



(a) Equal-size partitions (b) Unequal-size partitions

Jetzt Größe variabel lassen

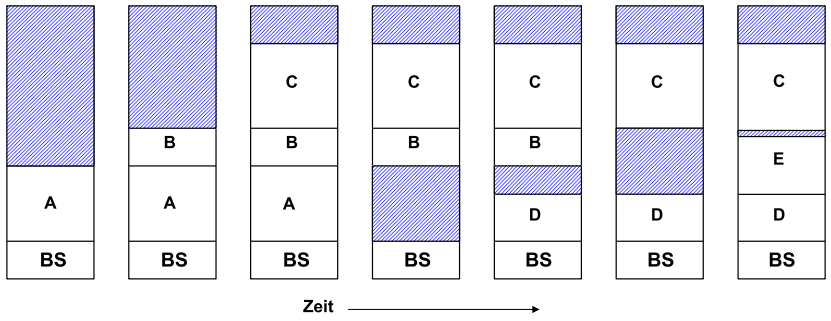
Bisher:

- Aufteilung in Partitionen beim Systemstart (für Platten: beim Formatieren) festgelegt
- Änderung der Partitionierung erfordert Neustart / Neuformatierung mit Verlust aller Informationen (alle Prozesse beenden bzw. alle Dateien löschen)

Jetzt:

- Partitionierung dynamisch, also keine feste Aufteilung bei Systemstart / Formatierung
- viel höhere Flexibilität (Größe/Anzahl der Prozesse),
- aber deutlich mehr Verwaltungsaufwand

Dynamische Partitionierung



Mit der Zeit entstehen immer mehr kleine Löcher zwischen Partitionen – **externe Fragmentierung**

Dafür fast keine interne Fragmentierung mehr

Wachsender Speicherbedarf (1/2)

Was passiert, wenn der Speicherbedarf eines Prozesses wächst?

- Ist neben der Partition ein freier Speicherbereich ⇒ Partition vergrößern
- Neue, ausreichend große Partition reservieren und den Inhalt dorthin verschieben
- Gibt es keine ausreichend große Partition ⇒ einen oder mehrere Prozesse auf Platte auslagern (**Swapping**).

Wachsender Speicherbedarf (2/2)

Was passiert, wenn der Speicherbedarf eines Prozesses wächst?

- Man kann auch dem Prozess von vornherein einen größeren Speicherbereich als angefordert zuweisen
  ⇒
  - interne Fragmentierung
  - Wenn auch dieser größere Bereich nicht ausreicht, muss wieder eines der vorher beschriebenen Verfahren angewandt werden

Verwaltung freier Bereiche

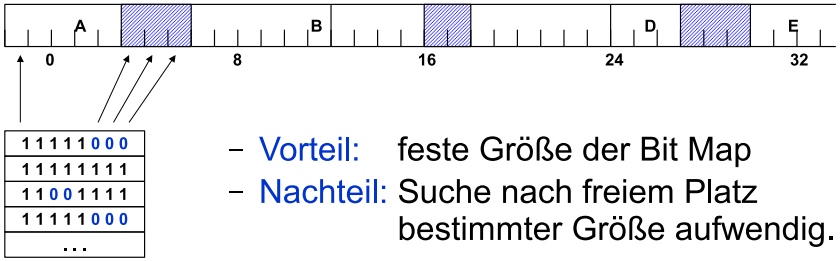
bei variablen Partitionen: komplizierter

zur Verwaltung freier / belegter Bereiche eine von zwei Datenstrukturen:

- Bit Maps
- Linked Lists

Bit Maps

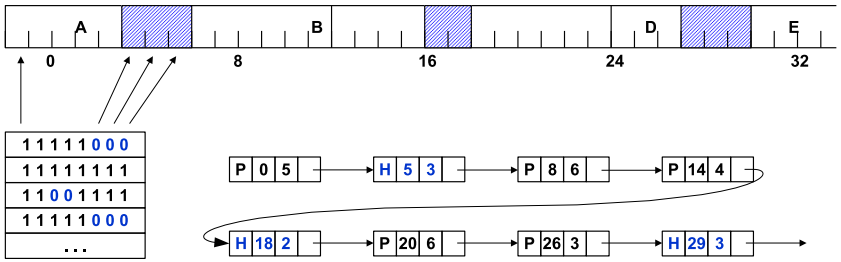
- Aufteilung des Speichers in Einheiten (einige Byte bis einige KByte)
- Ein Bit pro Einheit:
  - 0 – wenn die Einheit frei ist
  - 1 – wenn die Einheit belegt ist



- Vorteil: feste Größe der Bit Map
- Nachteil: Suche nach freiem Platz bestimmter Größe aufwendig.

Linked Lists

- Verkettete Liste von Beschreibungen der Speicherbereiche:
  - Von Prozess belegt (P) oder frei (H=hole)
  - Anfangsadresse und Länge des Bereichs
  - Zeiger auf nächsten Eintrag



Zuteilung freien Speichers

Mehrere Ansätze:

- **First-Fit**-Methode (schnell: erster, der passt)
- **Best-Fit**-Methode (optimiert)
- **Worst-Fit**-Methode
- **Quick-Fit**-Methode (beschleunigtes Best-Fit)
- **Buddy System**

First-Fit-Methode

- Der *erste genügend große* freie Speicherbereich wird zugeordnet
- schnellstes Verfahren, keine Prüfung des ganzen Speichers nötig
- Variante: Next-Fit (merkt sich zuletzt ausgewählte Position)

Best-Fit-Methode

- Der *kleinste ausreichende* freie Speicherbereich wird zugeordnet
- Aufwendiger, da die gesamte Liste bzw. Bitmap durchsucht werden muss
- Starke Fragmentierung des freien Speichers in viele kleine Bereiche

Worst-Fit-Methode

- Der *größte* freie Speicherbereich wird zugeteilt.
- Es bleiben verhältnismäßig große freie Bereiche übrig.

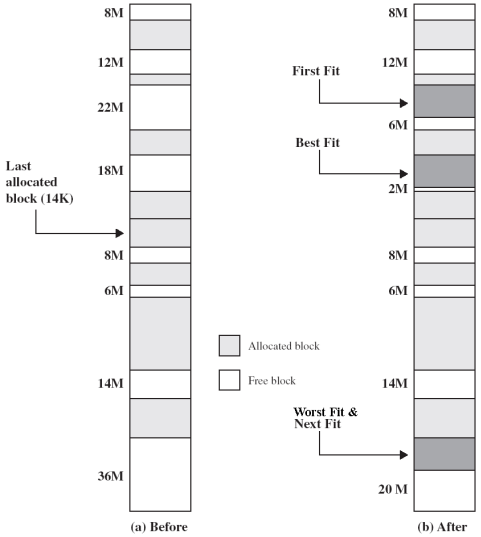
Quick-Fit-Methode

- Unterhalten von mehreren Listen freier Speicherbereiche für bestimmte Standard-größen (z. B. 4 KB, 8 KB, 12 KB etc.) führt zu einer schnellen Zuteilung.
- Bei Freigabe eines Bereiches muss dieser mit evtl. benachbarten freien Bereichen zusammengelegt werden. Dies bedeutet bei mehreren Listen einen erhöhten Aufwand.
- Spezielle Variante: Buddy System

Beispiele

Beispiel für eine Speicherbelegung vor (links) und nach (rechts) der Allokation eines 16-MByte-Blocks

Bild: Stallings



Buddy-System

- Separate Listen freier Bereiche der Größen 1, 2, 4, 8, 16 etc. KByte bis zur Speichergröße.
- Bei Freigabe eines Speicherbereichs muss nur eine der Listen durchsucht werden, um festzustellen, ob der Bereich mit einem anderen freien Bereich zusammengefasst werden kann.
- Blockgröße immer Zweierpotenz ist nicht sehr speichereffizient.

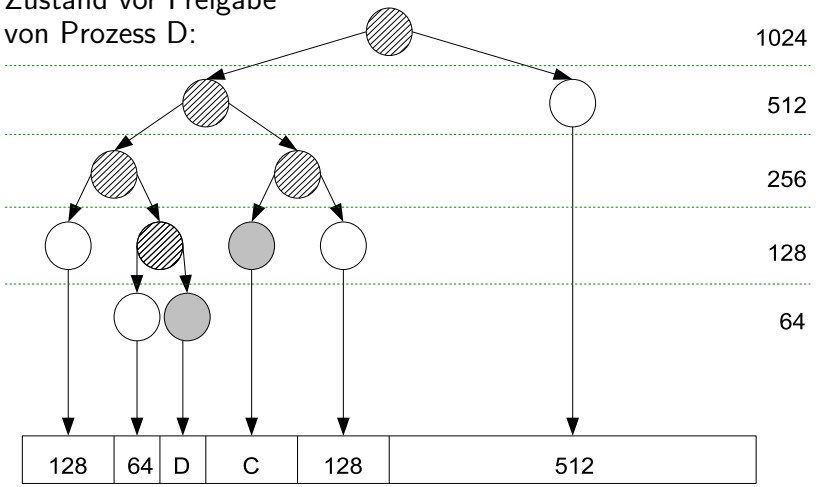
Buddy-System: Beispiel

Beispiel für Speicherzuordnung nach dem Buddy-System:

	0 – 128 K	128 K – 256K	256K – 384 K	384 K – 512K	512K – 640K	640K – 768K	768K – 896K	896K – 1M
Anfang	1024							
Anfrage 70	A	128		256				512
Anfrage 35	A	B	64	256				512
Anfrage 80	A	B	64	C	128			512
Freigabe A	128	B	64	C	128			512
Anfrage 60	128	B	D	C	128			512
Freigabe B	128	64	D	C	128			512
Freigabe D	256			C	128			512
Freigabe C	1024							

Buddy-System: Baum-Repräsentation

Zustand vor Freigabe von Prozess D:



Besonderheiten bei Speicherverwaltung: Code-Verschiebung und Speicherschutz

- Code-Verschiebung: Ein Programm muss an verschiedenen Stellen im Speicher laufen können
- Speicherschutz: Ein Programm darf nicht auf den Speicher-bereich eines anderen Programms zugreifen

Code-Verschiebung, Beispiel: Z80

Code in Z80-Assembler:

```
#base 0xA000                    ; Programmstart bei 0xA000

ld hl,data              A000    ld hl,0xA011    ; data
ld a,(hl)               A003    ld a,(hl)
call print              A004    call 0xA008     ; print
ret                     A007    ret

:print                          ; Funktion print
cmp a,' '               A008    cmp a,' '
jne ende                A00A    jne 0xA010      ; ende
call OS_PRINT           A00D    call OS_PRINT
:ende                           ; Label ende
ret                     A010    ret

:data                           ; Daten
char 'x'                A011    char 'x'
```


Code-Verschiebung, Probleme

Probleme mit nachträglicher Relokation:

- Was sind Adressen?
  - Sprung- und Funktionsaufruf-Adressen
  - Zugriff auf Datenbereiche
- Was sieht nur zufällig wie eine Adresse aus?
- Mehrfach indirekte Adressierung („lade den Wert, der in der Speicherstelle steht, deren Adresse hier steht“)

Besser: Lade-System plant Relokation von vornherein ein, merkt sich Stellen mit Adressen

Code-Verschiebung

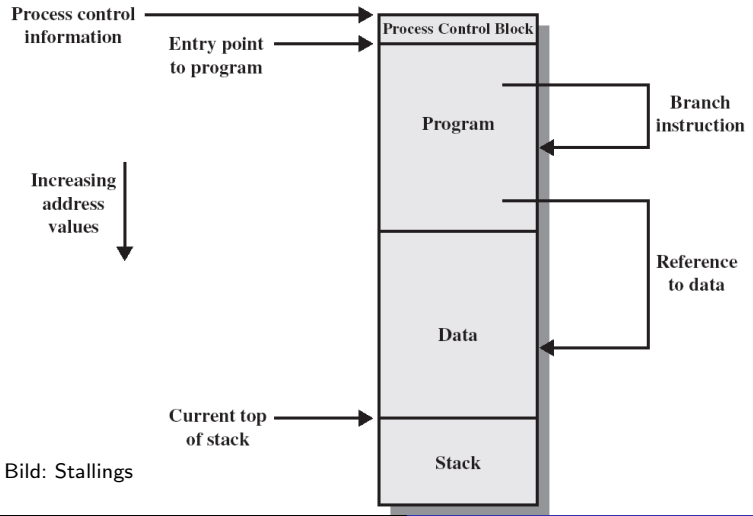


Bild: Stallings

Code-Verschiebung

Ein Programm muss an verschiedenen Stellen im Speicher laufen können. Zwei Möglichkeiten:

- Linker vermerkt, welche Code-Stellen absolute Adressen sind. Beim Laden des Programms werden diese Stellen entsprechend abgeändert
- Der Rechner hat ein spezielles Hardware-Register, ein sog. **Basisregister**
  Bei jeder Adressberechnung (zur Laufzeit) wird die Adresse im Basisregister zu der Adresse im Programm addiert.

Speicherschutz

Ein Programm darf nicht auf den Speicherbereich eines anderen Programms zugreifen.

Zwei Möglichkeiten:

- Schutzcode (vor jedem Speicherzugriff prüfen, ob Adresse gültig ist)
- Der Rechner hat ein spezielles Hardware-Register, ein sog. **Längenregister**.
  Durch Überprüfen des Längenregisters feststellen, ob die zugegriffene Adresse in der Partition des Programms liegt.

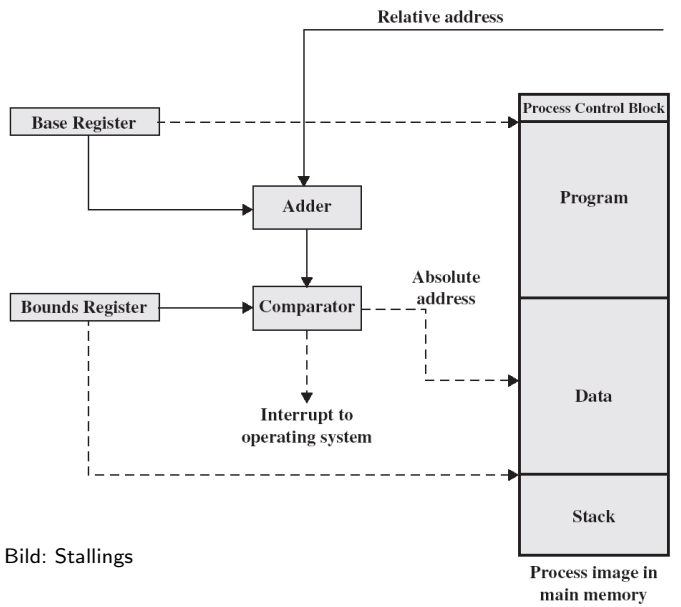
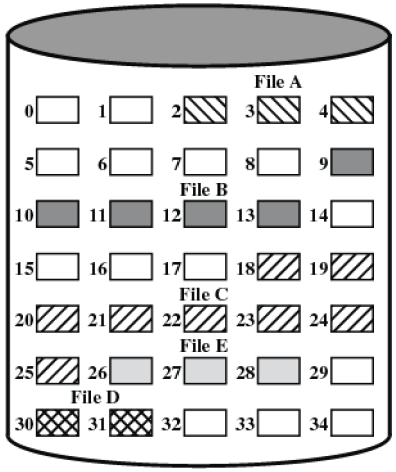


Bild: Stallings

. . . und bei Dateisystemen?

- Alle Verfahren und Konzepte auf Dateisysteme übertragbar
- z. B. Verwaltung freien Platzes bei variabler Partitionierung:
  - „kleinste Einheiten“: Blöcke
  - Bit Map (belegt/frei): Free Block Bitmap
- Dateisystemeintrag: Dateiname, Größe, Adresse des 1. Blocks (immer noch im Modell mit zus.-hgd. Speicherverwaltung!)

. . . und bei Dateisystemen?



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Bild: Stallings, Kapitel 12

. . . und bei Dateisystemen?

Was bedeuten hier „Code-Verschiebung“ und „Speicherschutz“?

- „Code-Verschiebung“: einfacher gestrickt; es gibt keine Sprünge in Dateiadressen. Bei Zugriff auf Datei nur relative Dateiadresse in absolute Plattenadresse umrechnen (Dateianfang + Offset)
- „Speicherschutz“: Nicht über das Ende der Datei-Partition hinaus (und damit evtl. in eine andere Datei) schreiben oder lesen

Vorschau

- Morgiges Praktikum: Einige Konzepte der Speicherverwaltung (RAM) auf Dateisysteme übertragen und in Python umsetzen
- Nächste Vorlesung: Verfahren der *nicht*-zusammenhängenden Speicherverwaltung