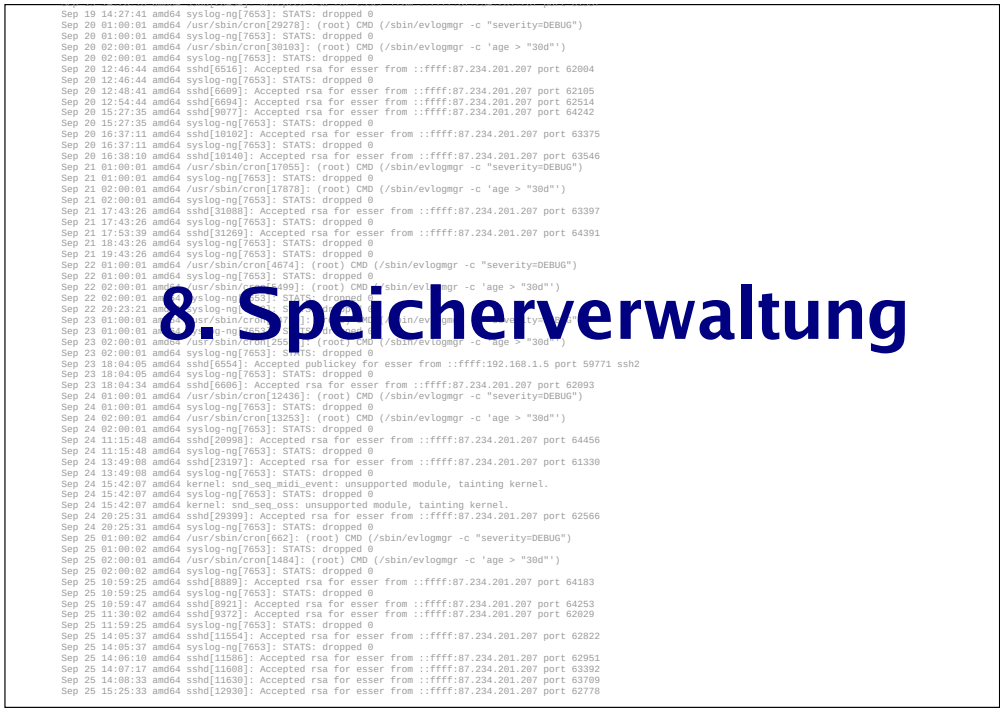
Betriebssysteme II

FH München – WS 2006/07

Hans-Georg Eßer

Zusammenfassung

/home/esser/Daten/Dozent/Folien/bs2-esser-13.odp



Gliederung

2. Prozesse und Threads
3. Interrupts
4. Scheduler

} Montag

5. Synchronisation
6. Interprozess-Kommunikation (IPC)
7. Deadlocks

} Mittwoch

8. Speicherverwaltung
9. Dateisysteme

} **heute**

Arten der Speicherverwaltung

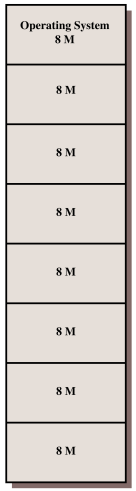
- Zusammenhängende Speicherzuteilung
  - Jede Anforderung eines Prozesses nach einer bestimmten Menge Speicher muss das BS durch zusammenhängenden (contiguous) Speicher erfüllen.
- Nicht zusammenhängende Speicherzuteilung
  - BS kann Speicheranforderung durch Zuweisung mehrerer kleiner Speicherbereiche erfüllen, die zusammen die geforderte Menge Speicher ergeben.
  - Wiederauffinden der verstreuten Speicherbereiche ist eine Aufgabe der Speicherverwaltung.

Multiprogramming

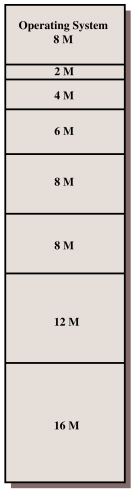
- Programme verbringen einen großen Teil ihrer Ausführungszeit mit Warten (auf I/O etc.).
- Auslagern (Swapping) auf Platte bei jedem Warten ist ineffizient.
- Lösung: mehrere Programme gleichzeitig im Speicher.
- Voraussetzung: verschiebbarer Code und Speicherschutz.

Feste Partitionen

- Aufteilung des Speichers in Partitionen fester (gleicher oder ungleicher) Größe.
- Zuweisung eines Programms zu einer freien Partition. Alternativen:
  - Erstes Programm, das in die freie Partition passt (eine Warteschlange)
  - FIFO für jede einzelne Partition (mehrere Warteschlangen)
  - Größtes Programm, das in die freie Partition passt



(a) Equal-size partitions

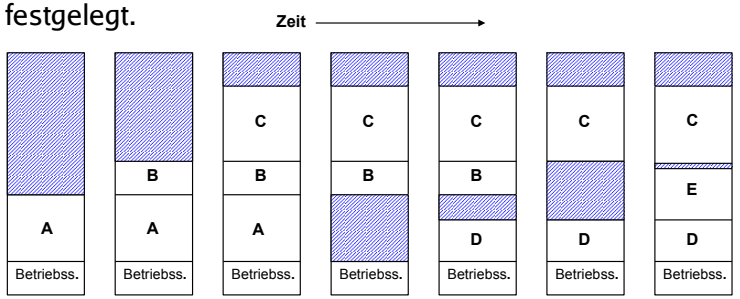


(b) Unequal-size partitions

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

Aufteilung in variable Partitionen (1)

- Anzahl und Größe der Partitionen werden dynamisch festgelegt.



- Es bleiben u. U. viele kleine freie Bereiche im Hauptspeicher (Löcher). Dies wird als externe Fragmentierung bezeichnet.
- Evtl. müssen diese Löcher durch Verschieben der Partitionen entfernt werden (memory compaction).

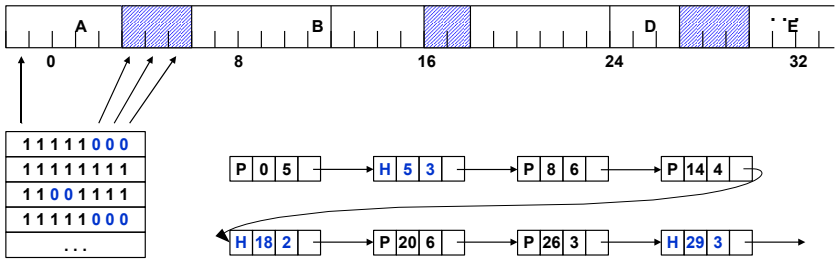
Aufteilung in variable Partitionen (2)

Was ist, wenn der Speicherbedarf eines Prozesses wächst?

- Wenn neben der Partition ein freier Speicherbereich ist, kann die Partition vergrößert werden.
- Es kann eine neue, ausreichend große Partition reserviert und der Inhalt dorthin verschoben werden.
- Wenn keine ausreichend große Partition zur Verfügung steht, müssen ein oder mehrere Prozesse auf Platte ausgelagert werden (Swapping).
- Es kann auch dem Prozess von vornherein ein größerer Speicherbereich zugewiesen werden, als angefordert.
  - interne Fragmentierung.
  - Wenn auch dieser größere Bereich nicht ausreicht, muss wieder eines der vorher beschriebenen Verfahren angewandt werden.

Welche Bereiche sind frei?

- **Bitmap:** Aufteilung des Speichers in Einheiten (einige Byte bis einige KByte); ein Bit pro Einheit: 0 / 1 für frei / belegt
- **Verkettete Liste** von Beschreibungen der Speicherbereiche:
  - Von Prozess belegt (P) oder frei (H=hole)
  - Anfangsadresse und Länge des Bereichs
  - Zeiger auf nächsten Eintrag

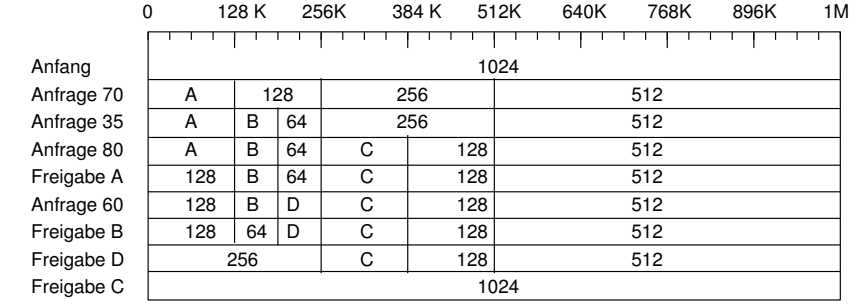


Zuteilung freien Speichers

- First-Fit-Methode: Der erste genügend große freie Speicherbereich wird zugeordnet.
- Best-Fit-Methode: Der kleinste ausreichende freie Speicherbereich wird zugeordnet. (Aufwendiger, da die gesamte Liste / 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: Listen freier Speicherbereiche für bestimmte Standardgrößen (z. B. 4 KB, 8 KB, 12 KB etc.) -> schnelle Zuteilung. Bei Freigabe den Bereich mit evtl. benachbarten freien Bereichen zusammenlegen
  - Spezielle Variante: Buddy System

Speicherverwaltung: 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.

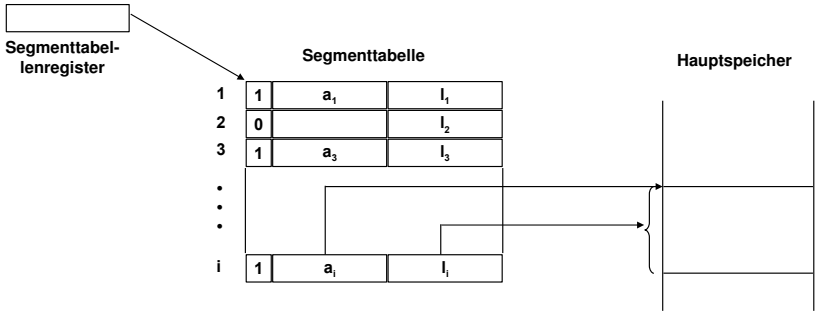


Segmentierung (1)

- Aufteilung von Programmen in mehrere Segmente, z. B. Codesegment, Datensegment(e), Stacksegment(e) etc.
- Jedes Segment hat linearen Adressraum von 0 bis Maximalwert.
- Adressangaben: Segmentangabe und relative Adresse
- Vorteile:
  - Für die kleineren Segmente findet man leichter freien Speicher.
  - Segmente können unabhängig voneinander wachsen.
  - Es müssen nicht alle Segmente gleichzeitig im Hauptspeicher sein (Swapping von Segmenten).
  - Segmente können unterschiedlich geschützt werden.
  - Unterscheidung von prozess-privaten Segmenten und solchen Segmenten, die von mehreren Prozessen gemeinsam benutzt werden können.

Segmentierung (2)

- Segmentierung muss von der Hardware unterstützt werden.
- Für jeden Prozess gibt es eine Segmenttabelle im Speicher, die für jedes Segment mindestens drei Angaben enthält:
  - Ist das Segment im Hauptspeicher?
  - Anfangsadresse des Segments im Hauptspeicher
  - Länge des Segments



Segmentierung (3)

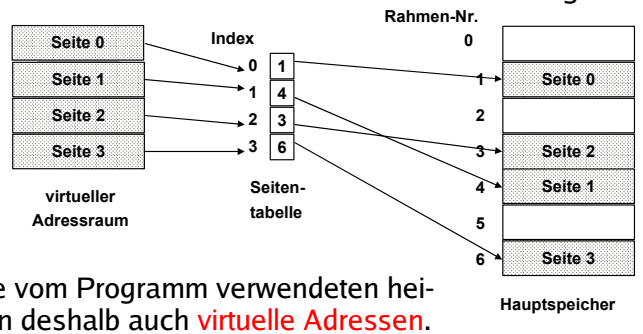
- Zugriff auf ein Segment, das nicht im Hauptspeicher ist
  -> Segment Fault: Betriebssystem muss das Segment erst in den Hauptspeicher laden.
- Schutz eines Segments:
  Die Einträge in der Segmenttabelle enthalten zusätzlich einen Schutzcode (z. B. für die Zugriffe Lesen, Schreiben, Ausführen), der vom Programmierer (oder Compiler/Linker) festgelegt wird.
- Sharing eines Segments im Hauptspeicher ist einfach:
  Man lässt die Einträge in den Segment-tabellen mehrerer Prozesse auf die gleiche Hauptspeicheradresse zeigen.
- Fragmentierung:
  Da jedes Segment im Hauptspeicher zusammenhängend abgelegt wird, tritt externe Fragmentierung auf.

Virtuelle Speicherverwaltung (Paging)

- Aufteilung des Adressraums in Seiten (pages) fester Größe und des Hauptspeichers in Seitenrahmen (page frames) gleicher Größe.
- Der lineare, zusammenhängende Adressraum eines Prozesses („virtueller“ Adressraum) wird auf beliebige, nicht zusammenhängende Seitenrahmen abgebildet.
- Eine einzige Liste freier Seitenrahmen verwalten
- Berechnung der physikalischen Speicheradresse aus der vom Programm angegebenen virtuellen Adresse
  - zur Laufzeit des Programms, transparent für das Programm,
  - muss von der Hardware unterstützt werden.
- Vorteile der virtuellen Speicherverwaltung:
  - Einfache Zuteilung von Hauptspeicher.
  - Keine externe Fragmentierung, geringe interne Fragmentierung.
  - Kein Aufwand für den Programmierer.

Virtueller Adressraum

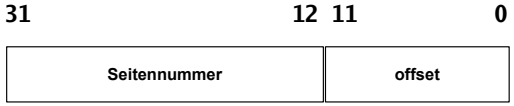
- Beim Paging wird der Zusammenhang zwischen Programmadresse und physikalischer Hauptspeicheradresse erst zur Laufzeit mit Hilfe der Seitentabellen hergestellt.



- Die vom Programm verwendeten hei-ßen deshalb auch virtuelle Adressen.
- Der virtuelle Adressraum eines Programms ist der lineare, zusammenhängende Adressraum, der dem Programm zur Verfügung steht.

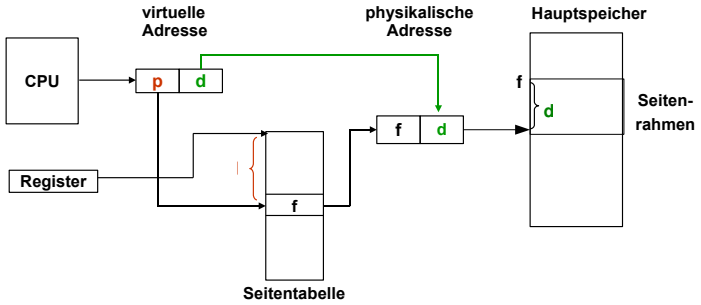
Adressübersetzung beim Paging (1)

- Programmadresse in zwei Teile aufteilen:
  Seitennummer + relative Adresse (Offset) in der Seite
  Beispiel: 32-bit-Adresse bei einer Seitengröße von 4096 (=2^12) Byte:



- Für jeden Prozess gibt es eine Seitentabelle (page table). Diese enthält für jede Prozess-Seite
  - eine Angabe, ob die Seite im Speicher ist,
  - die Nummer des Seitenrahmens im Hauptspeicher, der die Seite enthält.
- Ein spezielles Register enthält die Anfangsadresse der Seitentabelle für den aktuellen Prozess.
- Seitennummer als Index in die Seitentabelle verwenden.

Adressübersetzung beim Paging (2)



- Für jeden Hauptspeicherzugriff wird ein zusätzlicher Hauptspeicherzugriff auf die Seitentabelle benötigt. Dies muss durch Caches in der Hardware beschleunigt werden!
- Seite nicht im Speicher -> page fault (Seitenfehler) auslösen.

Virtueller Speicher allgemein (1)

- Mehr Prozesse können effektiv im Speicher gehalten werden --> bessere Systemauslastung
- Ein Prozess kann viel mehr Speicher anfordern als physikalisch verfügbar
- allgemeiner Vorgang:
  - Nur Teile des Prozesses befinden sich im physikalischen Speicher
  - falls Zugriff auf eine Adresse, die ausgelagert ist:
    - BS setzt den Prozess auf blockiert
    - BS setzt eine Disk-I/O-Leseanfrage ab
    - Nach Laden des fehlenden Stücks (Seite oder Segment) wird ein I/O-Interrupt abgesetzt
    - BS setzt Prozess zuletzt wieder in den Bereit-Zustand

Virtueller Speicher allgemein (2)

- „*thrashing*“ (siehe später): Prozessor verbringt die meiste Zeit mit Ein- und Auslagern von Prozessteilen statt mit der Ausführung von Prozessanweisungen
- Lokalitätsprinzip:
  - Zugriffe auf Daten und Programmcode häufig lokal gruppiert;
    --> Annahme gerechtfertigt, dass nur wenige Prozess-stücke während einer kurzen zeitlichen Periode gleichzeitig vorgehalten werden müssen

Translation Look-Aside Buffer (1)

- Translation Lookaside Buffer (TLB): schneller Hardware-Cache, mit den zuletzt benutzten Seitentabelleneinträgen
- Assoziativ-Speicher: bei Übersetzung einer Adresse wird deren Seitennummer gleichzeitig mit allen Einträgen des TLB verglichen.
- Treffer im TLB -> Speicherzugriff auf Seitentabelle unnötig
- Fehltreffer -> Zugriff auf die Seitentabelle (Alten Eintrag im TLB durch neuen ersetzen)
- Trefferquote (hit ratio) beeinflusst die durch-schnittliche Zeit einer Adressübersetzung.
- Lokalitätsprinzip -> auch bei kleinen TLBs hohe Trefferquoten

Translation Look-Aside Buffer (2)

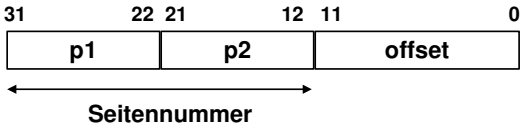
- Inhalt des TLB ist prozessspezifisch!
  Zwei Möglichkeiten:
  - Jeder Eintrag im TLB hat „valid bit“. Bei Prozesswechsel (Context Switch) gesamten TLB invalidieren.
  - Jeder Eintrag im TLB enthält Prozessidentifikation (PID), die mit der PID des zugreifenden Prozesses verglichen wird.

Invertierte Seitentabellen

- Bei großem virtuellen Speicher sehr viele Einträge in der Seitentabelle nötig, z.B. 2^{32} Byte Adressraum, 4 Kbyte/Seite
  -> über 1 Mio. Seiteneinträge, also Tabelle >4 MByte (je Prozess)
- Platz sparen durch invertierte Seitentabellen:
  - normal: ein Eintrag pro (virtueller) Seite mit Verweis auf den Seitenrahmen (im Hauptspeicher)
  - invertiert: ein Eintrag pro Seiten*rahmen* mit Verweis auf Tupel (Prozess-ID, virtuelle Seite)
- Problem: Suche zu Prozess *p* und seiner Seite *n* nach dem Eintrag (*p*,*n*) in der invertierten Tabelle -> langwierig
- Auch hier TLB einsetzen, um auf „meist genutzte“ Seiten schnell zugreifen zu können
- Bei TLB-Miss hilft aber nichts: Suchen...
- Andere Lösung für Problem der großen Seitentabellen: Mehrstufiges Paging

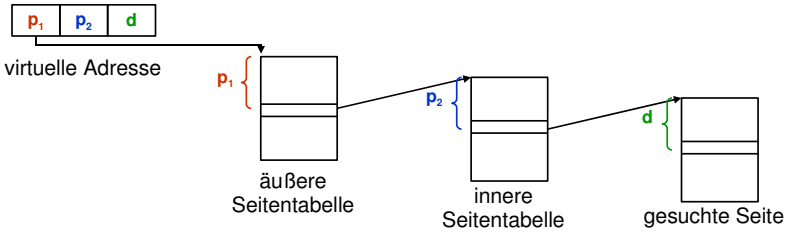
Mehrstufiges Paging (1)

- Zweistufiges Paging:
  - Seitennummer noch einmal unterteilen, z. B.:

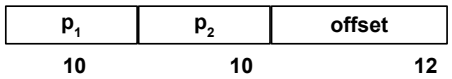


  - p_1: Index in äußere Seitentabelle, deren Einträge jeweils auf eine innere Seitentabelle zeigen
  - p_2: Index in eine der inneren Seitentabellen, deren Einträge auf Seitenrahmen im Speicher zeigen
  - Die inneren Seitentabellen müssen nicht alle speicherresident sein
- Analog dreistufiges Paging etc. implementieren

Mehrstufiges Paging (2)



- Größe der Seitentabellen:
  Beispiel:



  - Äußere Tabelle: 1024 (210) Einträge, die auf (bis zu) 1024 innere Tabellen zeigen, die wieder je 1024 Einträge haben.
  - 4 Byte pro Tabelleneintrag -> Größe der Tabelle = 1 Seite
  - nur so viele innere Seitentabellen verwenden, wie nötig.

Segmentierung mit Paging (1)

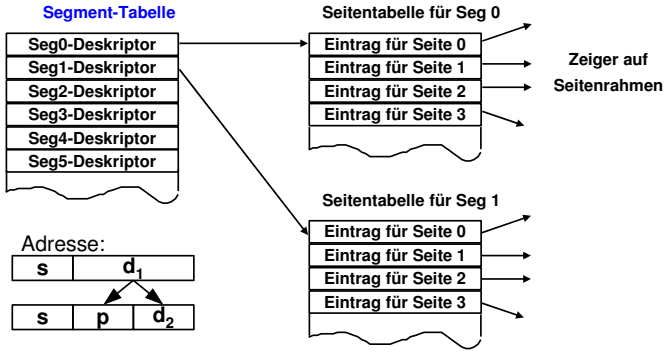
- Programme in Segmente unterteilen; nicht zusammenhängend, sondern mit Hilfe von Paging im Speicher ablegen.
- Programmierer gibt die Segment-Nummer und eine relative Adresse in diesem Segment an. Das Paging wird (transparent für das Programm) von der Hardware durchgeführt.

Zwei Realisierungsmöglichkeiten:

- Eine Segmenttabelle und pro Segment (pro Prozess) eine eigene Seitentabelle; Einträge in der Segmenttabelle zeigen auf die Seitentabelle des jeweiligen Segments
- Eine Segmenttabelle, eine einzige Seitentabelle (pro Prozess)
  - Einträge in der Segmenttabelle enthalten die Basisadresse des Segments in einem linearen (virtuellen) Adressraum.
  - relative Adresse im Segment zu dieser Basisadresse addieren. Resultierende lineare (virtuelle) Adresse mit Seitentabelle in eine physikalische Adresse übersetzen.

Segmentierung mit Paging (2)

- Eine eigene Seitentabelle pro Segment:



- Eine Adressangabe besteht aus Segmentnummer und linearer Adresse im Segment.
  Die Aufteilung der linearen Adresse in Seitennummer und Offset (für das Paging) ist transparent für das Programm.

Demand Paging

- Der Adressbereich eines Prozesses muss nicht vollständig im Hauptspeicher sein.
  - **Lokalitätsprinzip:** Prozess spricht in einer Zeitspanne nur relativ wenige, nahe beieinanderliegende Adressen an.
  - Teile des Programms werden bei einem bestimmten Ablauf evtl. nicht benötigt (Spezialfälle, Fehlerbehandlung etc.).
- Demand Paging bedeutet
  - dass eine Seite nur dann in den Speicher geladen wird, wenn der Prozess sie anspricht,
  - Seite kann auch wieder aus dem Speicher entfernt werden
- Vorteile von Demand Paging:
  - Der Adressbereich eines Prozesses kann größer sein als der physikalische Hauptspeicher.
  - Prozesse belegen weniger Platz im Hauptspeicher, somit können mehr Prozesse gleichzeitig aktiv sein.

Voraussetzungen für Demand Paging

- Jeder Eintrag in der Seitentabelle enthält ein valid bit, das angibt, ob die Seite im Speicher ist oder nicht.
- Wenn ein Prozess eine Seite anspricht, die nicht im Speicher ist, wird eine spezielle Exception ausgelöst, ein sog. page fault.
- Eine Betriebssystem-Routine, der page fault handler, lädt bei einem page fault die benötigte Seite in den Speicher.
- Falls kein freier Seitenrahmen im Speicher vorhanden ist, muss eine andere Seite ersetzt werden. Für die Auswahl der zu ersetzenden Seite muss eine Strategie implementiert werden.
- Die durch den page fault unterbrochene Instruktion muss erneut ausgeführt werden (können).

Seitenersetzung

- Wenn bei einem Page Fault kein freier Seitenrahmen zur Verfügung steht, muss das Betriebssystem einen frei machen.
- Algorithmus wählt nach einer Strategie diesen Rahmen aus.
- Falls die zu ersetzende Seite, seit sie zuletzt in den Speicher geholt wurde, verändert wurde, muss ihr aktueller Inhalt gesichert werden:
  - Ein modify bit (dirty bit) vermerkt, ob Seite verändert wurde.
  - veränderte Seite auf Platte sichern (Page-/Swap-Bereich).
- Eine unveränderte Seite kann später - bei Bedarf - wieder von der alten Stelle auf der Platte geladen werden.
- Im Seitentabelleneintrag für die ersetzte Seite valid bit löschen und merken, von wo die Seite wieder geladen werden kann.
- Seitenersetzungsstrategien: So wenig page faults wie möglich
- Zwei prinzipielle Arten von Seitenersetzungsstrategien: lokale Ersetzung vs. globale Ersetzung

Seitenersetzungsstrategien

Lokale Ersetzung: Es wird immer eine Seite desjenigen Prozesses ersetzt, der eine neue Seite anfordert.

- Die Zahl der Seiten, die ein Prozess im Speicher belegen kann, ist nach oben beschränkt. Die maximale Anzahl wird pro Prozess festgelegt (z. B. vom Systemverwalter) und kann die Laufzeit eines Prozesses stark beeinflussen.
- Ein Prozess, der viele Page Faults verursacht, z. B. weil er sich nicht an das Lokalitätsprinzip hält, beeinträchtigt nur sich selbst, nicht aber das Gesamtsystem.

Globale Ersetzung: Beliebige Seite im Speicher ersetzen.

- Prozesse nehmen sich gegenseitig Seiten weg.
- Ein Prozess, der viele Page Faults verursacht, erhält automatisch mehr Speicher. (Dies kann sowohl ein Vorteil als auch ein Nachteil sein.)

Optimale Strategie / FIFO

Optimale Strategie: Diejenige Seite ersetzen, auf die in Zukunft am längsten nicht zugegriffen wird.

- Vorteil: Diese Strategie verursacht die kleinste Zahl an Page Faults.
- Nachteil: Diese Strategie ist nicht implementierbar.

Die optimale Strategie kann modellhaft zur Bewertung anderer Strategien benutzt werden.

First In, First Out (FIFO): Die Seite ersetzen, die schon am längsten im Speicher ist.

- Vorteil: Sehr einfach zu implementieren:
- Nachteil: Die ersetzte Seite kann in dauernder Benutzung sein und gleich wieder angefordert werden.

Least Recently Used (LRU)

Die Seite ersetzen, die am längsten nicht benutzt worden ist.
- Vorteil: In der Regel weniger Page Faults als FIFO.
- Nachteil: Aufwändige Implementierung.

Zwei mögliche Implementierungen:
- Implementierung mit Zähler:
  - Systemweiten Zähler bei jedem Speicherzugriff inkrementieren.
  - Aktuellen Wert des Zählers in einem Feld in der Datenstruktur vermerken, welche die angesprochene Seite beschreibt.
  - Seite mit dem kleinsten Zählerwert ersetzen.
- Implementierung mit verketteter Liste:
  - Eine verkettete Liste enthält alle Seiten.
  - Bei jedem Speicherzugriff wird die angesprochene Seite an den Anfang der Liste gebracht. (Liste durchsuchen und Reihenfolge ändern, also Zeiger umsetzen!)
  - Die Seite am Ende der Liste wird ersetzt.

Referenz-Bits

- Jeder Seitentabelleneintrag kann ein Referenz-Bit enthalten
  - das bei einem Zugriff auf die Seite gesetzt wird (Hardware),
  - das nach bestimmten Kriterien gelöscht wird (Software).
- Ein Referenz-Bit
  - liefert die Information, ob auf eine Seite seit dem letzten Löschen des Bits zugegriffen wurde,
  - sagt nichts über den Zeitpunkt des Zugriffs auf eine Seite aus,
  - sagt nichts über die Reihenfolge der Zugriffe auf mehrere Seiten aus.
- Mit Referenz-Bits kann man weitere Seitenersetzungsstrategien implementieren, z. B.
  - Modifikationen von LRU, die weniger aufwändig sind.
  - Second-Chance-Algorithmus, eine Verbesserung der FIFO-Strategie.

Memory Mapped Files

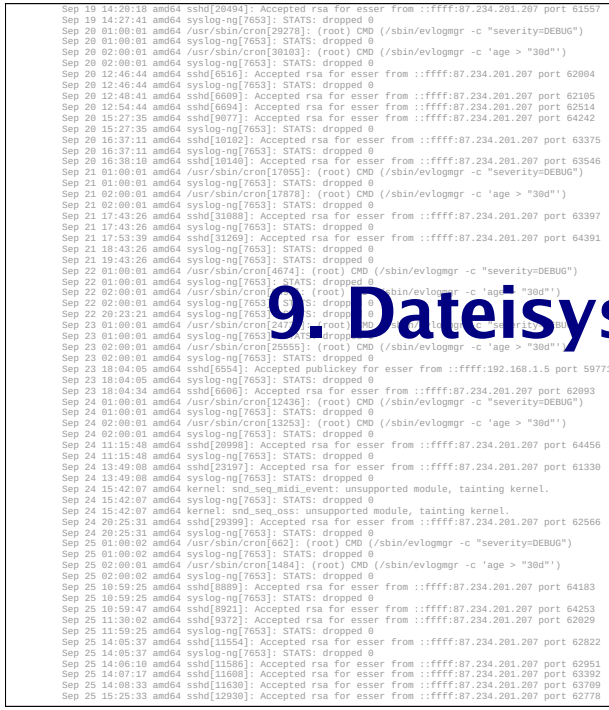
- Idee: Inhalt einer Datei als Speicher einblenden
- Zugriff auf Datei anschließend über Variablen / Speicheradresse möglich,
  z. B. Abbildung einer ASCII-Datei auf
  - String-Variable (Python)
  - char* -Variable (C)
- Auslesen und Verändern der Datei durch Manipulieren der Speicherstellen
- Standard-POSIX-Funktion `mmap()`

Swapping (1)

- Wichtig: zwei Bedeutungen von „Swapping“
  - **Eigentlich:** komplettes Auslagern eines Prozesses (oder bei Segmentierung: eines ganzen Segments) auf die Festplatte (to swap: vertauschen; ein Prozess geht aus dem Hauptspeicher, damit ein anderer herein kommen kann)
  - **Allgemeiner:** Auslagern von Teilen eines Prozesses (Paging: einzelne Seiten) auf die Platte
- Definition: Swapping ist die zeitweise Auslagerung aller von einem Prozess benutzten Speicherseiten (oder zumindest kompletter Segmente) auf einen Hintergrundspeicher (Platte), um z. B. bei zu wenig freiem Hauptspeicher Platz zu schaffen.
- Bei zusammenhängender Speicherzuteilung
  - ist Swapping die einzige Möglichkeit, mehr Programme gleichzeitig auszuführen, als im Hauptspeicher Platz haben,
  - müssen Speicherbereiche eines Prozesses, die dynamisch wachsen, u. U. ausgelagert werden.

Swapping (2)

- Kriterien für die Auslagerung: z. B. Prozesszustand, -priorität, -größe (im RAM); Zeit, die der Prozess im Hauptspeicher war
- Zuteilung und Verwaltung des Platzes im Swapbereich mit Verfahren der zusammenhängenden Speicherverwaltung.
- Swapping ist ein Vorläufer von Paging
- Da Prozesse immer „ganz oder gar nicht“ im Speicher sind, lassen Swapping-basierte BS keine Prozesse zu, deren Speicherbedarf größer als der Hauptspeicher ist
- Linux und Windows: kein Swapping, sondern Paging (nur aus Traditionsgründen heißt es oft „Swap-Partition“, „Swap-Datei“ etc.)
- sehr alte Unix-Versionen: Swapping



9. Dateisysteme

Virtuelles Dateisystem – VFS (1)

Zwei Schichten einführen

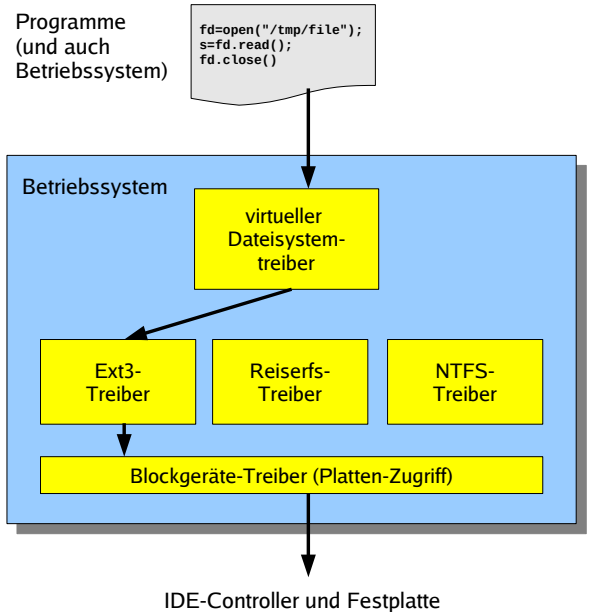
- VFS-Treiber stellt High-Level-Dateioperationen bereit (create, delete, rename, open, close, read, write, seek, link, ...)
- Kommunikation aller Programme (und auch des BS selbst) nur mit dem VFS-Treiber
- VFS-Treiber leitet Anfragen an Spezialtreiber für die Dateisysteme weiter
- Spezialtreiber beherrschen einzelne FS
- VFS-Standardfunktionen: create, delete, open, close, read, write, seek, append, get/set attrib, rename etc.
- Das VFS kennt die Datei-Attribute und -Operationen, die für das Betriebssystem relevant sind

VFS (2)

VFS-Treiber
- kennt abstrakte Datei-eigenschaften
- weiß, welchen speziellen Dateisystem-Treiber er braucht

Dateisystem-Treiber
- kennt das jeweilige Dateisystem-Format (weiß nichts von HW)

Blockgeräte-Treiber
- kann mit der Hardware sprechen



Linux VFS

Vier elementare Konzepte

- **Datei:** Folge von Bytes, keine „Dateitypen“ (aus Sicht des Dateisystems)
- **Verzeichnis:** Spezialdatei mit Informationen über Dateien im Verzeichnis
- **Inode:** Index node, enthält die Datei-Metadaten
- **Mount-Punkt:** Einbinden eines Dateisystems in den Verzeichnisbaum; der Mount-Punkt ist die Wurzel des Dateisystems

Linux VFS
Standard-Funktionen

- **fd = open (filename, flags)** Datei öffnen / erzeugen
- **close (fd)** offene Datei schließen
- **link ()** erzeugt neuen Verweis auf eine Datei
- **lseek (fd, offset, Art)** springt an eine andere Stelle in der Datei
- **read (fd, buffer, count); write (fd, buffer, count)**
- **stat (filename, status), fstat (fd, status)**
  Informationen über Datei abrufen
- **unlink (name)** Eintrag in einem Verzeichnis löschen
- **rename (oldpath, newpath)**
- **mmap (start, laenge, ..., fd, offset)**
  Datei ab Position *offset* mit Länge *laenge* in den Hauptspeicher einblenden

Linux-Dateiattribute (1)

- Besitzer (user) und Gruppe (group)
- Lese- (**r**), Schreib- (**w**) und Ausführrechte (**x**) für Besitzer (**u**), Gruppe (**g**) und sonstige Benutzer (**o**, others)
- ergibt 9 Zugriffsrechte; typische Notation:
  `-rwxrwxrwx` Besitzer, Gruppe, sonstige
- Anwender können zu verschiedenen Gruppen gehören
- numerische Rechte:
  - Lesen: 4 (2^2), Schreiben: 2 (2^1), Ausführen: 1 (2^0)
  - aufaddieren, z. B.: Lesen/Schreiben: 4+2=6
- für Benutzer, Gruppe und Sonstige: nnn, z. B. **640**:
  - Benutzer: 6 = lesen + schreiben (nicht ausführen)
  - Gruppe: 4 = lesen (nicht schreiben, nicht ausführen)
  - Sonstige: 0 = nichts

Linux-Dateiattribute (2)

- Erweiterte Attribute (nur ext2, ext3)
  - append only (**a**): auf Datei darf nur im Append-Modus geschrieben werden
  - auto-compression (**c**): Datei wird automatisch komprimiert
  - immutable (**i**): Datei darf nicht verändert werden
  - secure deletion (**s**): Wird diese Datei gelöscht, wird der Inhalt vorher mit 0-Bytes überschrieben
  - undeletion (**u**): Wird die Datei gelöscht, kann das rückgängig gemacht werden

Sekundärspeicher-Management (1)

- Platz für Dateien zuordnen (allozieren)
- Platz verwalten, der für Allokationen zur Verfügung steht
- **Pre-Allocation**
  - muss maximale Dateigröße schon bei Allokation kennen
  - Es ist schwer, verlässlich die Maximalgröße einer Datei vorherzusagen
  - Tendenz dazu, die Dateigröße zu überschätzen, um nicht in Platznot zu geraten
- **Zusammenhängende Allokation**
  - Beim Erzeugen einer Datei wird eine bestimmte Menge Blöcke alloziert
  - Ein einziger Eintrag in FAT: Anfang und Länge der Datei
  - Externe Fragmentierung (-> Dateisystem defragmentieren)

Sekundärspeicher-Management (2)

- **Schlangen-Allokation**
  - Allokation auf Basis individueller Blöcke
  - Jeder Block enthält einen Zeiger zum nächsten Block in der Schlange
  - Ein einziger Eintrag in der Zuordnungstabelle
    - Anfangsblock und Dateilänge
  - Keine externe Fragmentierung
  - optimal für sequentielle Dateien
  - Lokalitätsprinzip wird nicht berücksichtigt
  - Weit verteilte Blöcke zusammenfügen: **Konsolidierung**

Sekundärspeicher-Management (3)

- **Index-Allokation**
  - Dateizuordnungstabelle enthält für jede Datei einen separaten 1-Level-Index
  - Index hat für jede „Portion“, die der Datei zugeordnet wurde, einen Eintrag
    (Portionen sind die kleinsten Einheiten, die alloziert werden, von einzelnen Blöcken bis zur kompletten Datei)
  - In Dateizuordnungstabelle steht die Nummer des Blocks, der den Index enthält

Sekundärspeicher-Management (4)

Verwaltung des freien Plattenplatzes

- Platz, der zur Zeit nicht verwendet wird, verwalten
- Neben Dateizuordnungstabelle noch **Plattenzuordnungstabelle** mit Informationen über Freiräume
- **Bit-Table:** Bit-Vektor, in dem jedes Bit einen Block repräsentiert;
  0 = freier Block 1 = belegter Block
  (vgl. Bit-Vektoren für Verwaltung von freiem Hauptspeicher)
  -> Bit-Table passt evtl. komplett ins RAM (effizient)
- **Free Block List:** Blöcke werden durch nummeriert; eine Liste enthält die Nummern aller freien Blöcke
  - Liste zu groß, um sie komplett im Speicher zu halten
  - Abhilfe: Liste als FIFO-Warteschlange betrachten

Ext2/Ext3: Features

- variable Blockgrößen (kann je nach Einstellung gut mit vielen kleinen oder wenigen großen Dateien umgehen)
- Schnelle symbolische Links (Link-Ziel direkt im Inode speichern, wenn die Adresse kurz ist)
- Spezialattribute (die über die üblichen Unix-Dateiattribute hinaus gehen), z. B. *immutable*-Flag
- Extended Attributes, z. B. ACLs
- Kompatibilitäts-Bitmap für Weiterentwicklungen (*read/write*-, *read-only*- und *incompatible*-Bits)
- regelmäßige Überprüfung des Dateisystems (Mount-Count, Zeit seit der letzten Überprüfung)
- sicheres Löschen (durch Überschreiben der Daten)
- nur Ext3: Journaling

Ext2/Ext3: Aufbau (1)

- Aufbau: Partition in Boot-Sektor und **Blockgruppen** unterteilt
- Jede Blockgruppe enthält eine Kopie des **Superblocks** (mit Verwaltungsinformationen des ganzen Dateisystems)
- **Gruppen-Deskriptor:** Zustand der Block-gruppe (u.a. freie Blöcke und freie Inodes)
  (variable Länge)

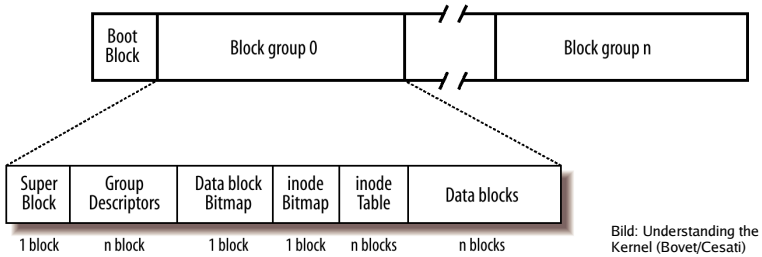


Bild: Understanding the Linux Kernel (Bovet/Cesati)

Ext2/Ext3: Aufbau (2)

- **Datenblock-Bitmap:** Für jeden Datenblock ein Bit (frei/nicht frei)
- **Inode-Bitmap:** Für jeden Inode ein Bit: benutzt/nicht benutzt
- **Inode-Tabelle:** Hier liegen alle Inodes der Blockgruppe (variable Länge)
- **Datenblöcke:** die eigentlichen Nutzdaten, also Dateien und Verzeichnisse
- Metainformationen in jeder Blockgruppe (Idee: Schutz durch Redundanz; kurze Entfernungen Metadaten/Nutzdaten)
- Tatsächliche Umsetzung anders:
  - Kernel hält im RAM Kopie des Superblocks, den es nur bei *fsck*-Aufrufen auf die Superblöcke verteilt
  - Spätere Ext2-Versionen: *Sparse-Superblock*-Option. Superblöcke nur in Gruppen 0, 1, 3^i, 5^i, 7^i (i>1)

Dateigrößen (1)

- Beim Erstellen des Dateisystems unterschiedliche Blockgrößen möglich:
  - 1024-Byte-Blöcke -> max. Größe 16 GByte
  - 2048-Byte-Blöcke -> max. Größe 256 GByte
  - 4096-Byte-Blöcke -> max. Größe 4096 GByte (4 TB)

  (x 2 zwischen den Zeilen, x 16 zwischen den Spalten)
- Bei großen Dateien: Speichern der Blocknummern in Indirektionsblöcken (bis zu 3 Stufen):
  Verdoppeln der Blockgröße = Ver-16-fachen (2^4) der Block-nummern: Faktor 2^3 aus Indirektion, Faktor 2 aus Blockgröße

Dateigrößen (2): Mehrfache Indirektion

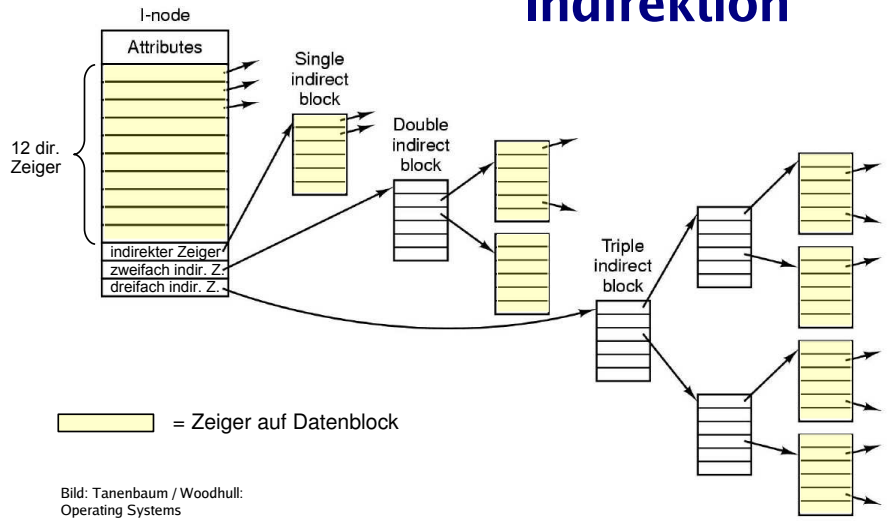


Bild: Tanenbaum / Woodhull: Operating Systems

Journaling (1)

- Transaktionskonzept ursprünglich von Datenbanken
- Journaling-Mechanismus für Dateisysteme vereinfacht (Performance und Implementationskomplexität).
- JFS zerlegt Dateisystemmodifikationen in **atomare Operationen** und gruppiert diese zu **Transaktionen**.
- Änderungen werden in **Journal** geloggt

Die Folien zum Journaling basieren auf folgendem Foliensatz:
Mengjun Xie, „The Ext3 File System", http://www.cs.wm.edu/~kearns/780S04/slides/ext3.pdf

Journaling (2)

- Alle Änderungen auf einem JFS in zwei Stufen durchführen:

1. Commit
   - Einzelne / mehrere Disk-Updates bilden eine Transaktion
   - Transaktion *ins Journal* (nicht ins FS) schreiben
   - Erst wenn ein „commit block" geschrieben wurde, ist die Transaktion abgeschlossen
   - Journal-Blöcke erst recyceln, wenn alle Daten im Dateisystem stehen (sync)
2. Check-point
   - Transaktion aus dem Journal ins Dateisystem übertragen (flush)
   - Journal-Speicherplatz freigeben

Journaling (3)

- Da Journaling teuer ist, schreiben die meisten JFS nur (Änderungen von) Metadaten ins Journal.
- Journaling sorgt für Konsistenz:
  - Commit-Phase ist atomar. Neue Daten werden ins Journal „committed", alte stehen noch im Dateisystem.
  - Nach einem Absturz kann fsck zwischen zwei Fällen unterscheiden:
    - Fehler trat vor Commit auf: Änderung geht zwar verloren (wird verworfen), aber Dateisystem ist konsistent, weil noch keine Flush-Operation gestartet wurde.
    - Fehler trat nach Commit auf: Änderung ist gültig und kann angewendet werden (flush).

Journaling (4)

- (Konsistenz ...)
  - Ext3 garantiert Konsistenz nur auf System-Call- Ebene. Darum gibt es für Operationen, die mehrere System Calls benötigen, keine Konsistenzgarantie.
  - Die meisten JFS legen keine Journal-Einträge für Operationen auf Dateidaten-Blöcken an
    ➞ Damit gibt es keinen Schutz vor Korruption einzelner Dateien (nur die Metadaten sind geschützt).