Zusammenfassung einiger Themen der SP-Vorlesung WS2016

July 14, 2019

Disclaimer:
Für Richtigkeit und Vollständigkeit wird keine Garantie gegeben. Die Zusammenfassung kann und soll die Folien und Literatur nicht ersetzen, sie ist lediglich als grobe Übersicht gedacht. Aus diesem Grund wurden einige Themen und viele Details weggelassen. Grundlagen, welche in anderen Vorlesungen besprochen wurden, wurden ebenfalls teilweise weggelassen.

Contents

1 Speichervirtualisierung

Abbildung von logischem Adressraum auf möglicherweise kleineren realen Adressraum mithilfe einer MMU.
Jeder virtuelle Adressraum gehört zu einem Prozessexemplar.

1.1 Ladestrategien

Rahmen für benötigte Seiten (entweder *on demand* durch einem trap (Pagefault) oder *anticipatory*) müssen verfügbar sein. Eventuell müssen andere Seiten verdrängt werden.

1.2 Ersetzungsstrategien

resident set:
Die im Hauptspeicher vorliegenden Seiten eines Prozessexemplars.

- Wenn Prozessen zu wenige Seiten im Hauptspeicher haben dürfen, erhöht sich die Rate von Pagefaults.
- Bei viel zu vielen Seiten sinkt die Rate nur unwesentlich
- Je weniger residente Seiten zugelassen werden, desto höher der Grad an Mehrprogrammbetrieb
- Befehlssatzebene legt Mindestgröße fest

Falls kein leerer Seitenrahmen verfügbar ist, wenn ein neuer geladen werden muss, muss **lokal** oder **global** nach einem Rahmen gesucht werden, der ersetzt werden kann. Dieser muss eventuell ausgelagert werden.

lokal: Gesucht wird unter den residenten Seiten des Prozesses der den Pagefault verursacht hat. Seitenfehler sind deterministisch.

global: Es werden alle Seitenrahmen durchsucht, Seitenfehler sind "zufällig".

Falls die Größe des resident set fest ist, ist eine globale Ersetzung nicht möglich. Die Größe des resident set eines Prozesses wird bei der Prozesserzeugung festgelegt.

Referenzfolge: Folge adressierter Adressen, wegen *Lokalitätsprinzip* meist stetig.

1.3 Strategien

Die Effektivität der Strategien hängt von der Referenzfolge des Prozesses ab.

1.3.1 FIFO

Zuerst eingelagerte Seite wird ersetzt. Mehr Seitenrahmen führen gegebenfalls zu mehr Pagefaults.

1.3.2 LFU (least frequently used)

Referenzierung von Seiten muss gezählt werden. (Anscheinend gibt es ein MFU, warum man aber die am häufigsten benutzte Seite auslagern soll..?)

1.3.3 LRU (least recently used)

Implementierung entweder durch Zeitstempel, Stack (neue Seiten obendrauf), Schieberegister oder mit Statusbits abschätzbar.

Alterungsstruktur (Schieberegister)

- jeder Seitendeskriptor hat ein *aging register* (meist Software)
- es wird periodisch geprüft ob eine Seite referenziert wurde, wenn nicht wird dies mit dem aging register vermerkt
- Je inaktiver die Seite, desto höher die Ersetzungspriorität

zweite Chance

- FIFO-Struktur mit Statusbit (*use bit*)
- Das Statusbit wird bei Referenzierung einer Seite gesetzt
- falls bei der periodischen Prüfung das Bit gesetzt ist, wird es zurückgesetzt, andernfalls kann die Seite ersetzt werden

Es wird also die zuletzt eingelagerte, *unbenutzte* Seite ersetzt. Falls die Statusbits aller Seiten gesetzt sind → FIFO.

dritte Chance Zusätzliches Modifikationsbit (*dirty bit*) wird bei Schreibzugriffen gesetzt und bei der Auswahl einer zu ersetzenden Seite mit geprüft.

use	dirty	Bedeutung	Entscheidung
0	0	ungenutzt	beste Wahl
0	1	beschrieben	keine schlechte Wahl
1	0	gelesen	keine gute Wahl
1	1	gelesen und beschrieben	schlechteste Wahl

Es wird eventuell zwei mal durch die FIFO-Liste iteriert.

1.4 Seitenflattern (thrashing)

Ein-/Auslagerung von Seiten dominiert Systemaktivität.

- falsche Seiten wurden ausgelagert und müssen sofort wieder eingelagert werden
- entsteht möglicherweise durch globale Seitenersetzung, wenn die Menge der residenten Seiten von Prozessen zu klein ist oder durch zu hohen Grad an Mehrprogrammbetrieb
- kann von alleine wieder verschwinden

1.5 Freiseitenpuffer (page buffering)

Reserve "ungebundender" Seitenrahmen, die logisch abwesend aber noch im Hauptspeicher sind.

- arbeitet nach FIFO-Prinzip, mit zusätzlichem Cache potentiell zu ersetzender Seiten
- eine Liste für modifizierte und eine für unmodifizierte Seiten
- modifizierte Seiten müssen vor dem Ersetzen ausgelagert werden
- falls eine Seite gebraucht wird, welche im Freiseitenpuffer ist, wird das present bit wieder gesetzt
- andernfalls wird aus einer der beiden Listen die lokal zuerst eingelagerte am längsten ungenutzte Seite ersetzt
- mit bestimmten Schwellenwerten kann entschieden werden ob Seiten brach liegen und dann andere Seiten vorausladen

Definitionen

- **working set (WS):**
  Menge der zuletzt referenzierten Seiten eines Prozesses, Teilmenge seiner residenten Seiten
- **Arbeitsmenge**:
  Menge der Seiten die der Prozess mindestens braucht für seine effiziente Ausführung

//Rechnungen die nicht allzu wichtig aussehen In der Zusammenfassung steht, dass der Freiseitenpuffer und das WS lokale Verfahren sind.

2 Speicher

- **Primärspeicher:**
  $\leq$ dutzende Sekunden, zum Beispiel Register oder Haupt-/Arbeitsspeicher
- **Sekundärspeicher:**
  im Mittel 5 Jahre, Ablagespeicher
- **Tertiärspeicher:**
  $\geq$ 10 Jahre, Festplatten-/Magnetbandarchive, DVDs ...

Je größer die Kapazizät und Zeitspanne desto langsamer und billiger.

- **logischer Adressraum:**
  Adressen von einem Programm. Keine Lücken, kann größer als der reale Adressraum sein
- **virtueller Adressraum:**
  Gleich wie logischer Adressraum, jedoch müssen nicht alle Adressen im Hauptspeicher vorliegen
- **realer Adressraum:**
  Durch Prozessor definierte Adressen, kann Lücken aufweisen

2.1 Speicherverwaltung

Zuteilung wird durch Maschinenprogramm und Betriebssystem gesteuert (Heap, Hauptspeicher). Das Maschinenprogramm verwaltet seinen Adressraum lokal selbst, zum Beispiel mit *malloc/free*. Das Betriebssystem verwaltet den ganzen Haupt-/Arbeitsspeicher global (*sbrk/mmap*). Verschiedene Zerlegungsgrade (Granularität) von Speicher:

- **Wort**:
  Vielfache eines Bytes
- **Zelle**:
  Platzierungseinheit für Halden- und Hauptspeicher, wenige Byte (Zweierpotenzen)
- **Kachel(Page?)**:
  Größe in welche der Arbeitsspeicher aufgeteilt ist, 512B, 4KiB etc.

Vielfache dieser Zerlegungen werden zu Segmenten zusammengefasst,
Segmente bilden Schutzeinheiten im logischen Adressraum.
Speicher kann mit Segmenten und/oder Pages verwaltet werden.

2.1.1 Segmentierung

- variable Größe
- Attribute gespeichert im Segmentdeskriptor: Granularität, Adresse, Länge etc.
- mithilfe einer Löcherliste (*hole list*) wird freier Speicher verwaltet
- die Liste ist nach Größen oder Adressen sortiert, Ausführlicher in *Kapitel 7.1.3 Lineare Lochlisten*
- Ziel ist Verschnitt möglichst klein zu halten (Löcher verschmelzen, bei Bedarf möglichst nicht allzu viel Platz verschwenden)

2.1.2 Paging

- Seiten (*pages)* haben feste Größe
- Seitenrahmen (*page frames*) haben gleiche Größe, die Rahmen gehören zum realen Adressraum (Ein Rahmen ist Platz im Speicher, an den eine Seite passt)
- eine oder mehrere Seitentabellen (*page table*) für Seitendeskriptoren (*page descriptor*), welche ein *present bit*, andere Bits für Metainformationen und entsprechender Adresse im Hauptspeicher (eingelagert) oder Blocknummer im Ablagespeicher (ausgelagert)
- jede logische/virtuelle Adresse besteht aus Seitennummer und *offset* innerhalb der Seite
- Seitennummern werden mithilfe der page tables auf page frames abgebildet ("die reale Adresse wird herausgefunden")

Beispiel: Logische Adresse 0x*babe* in reale Adresse und Offset innerhalb der Seite umrechnen, bei 2KiB Seitengröße.
2KiB = 2^{11}, die rechten 11 Bits der logische Adresse gehören zum Offset, der Rest zur realen Adresse.

$$0\text{x}babe \rightarrow b - 1010 - be \rightarrow \underbrace{b-1}_{5\ Bits} | \underbrace{010-be}_{11\ Bits} \rightarrow 10111|0\text{x}2be \rightarrow 0\text{x}17|0\text{x}2be$$

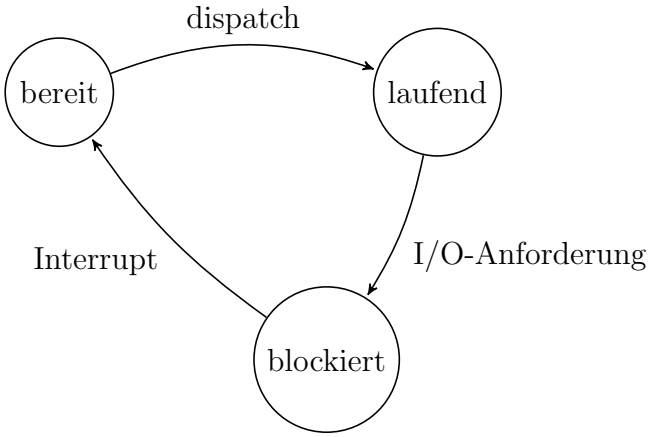
Die Page liegt also an Position 0x17 und der Offset innerhalb der page ist 0x2*be*.

3 Prozessvewaltung Einplanungsgrundlagen

- Einplanungseinheit ist der *Thread* (Faden)
- ein Thread ist laufbereit wenn alle von ihm gebrauchten *Betriebsmittel* (CPU, Speicher...) zur Verfügung stehen
- die Einplanung von Threads ist Zeit- oder Betriebsmittelorientiert
- *CPU burst* (Rechenstoß) aktive Phase, Thread hat alle nötigen Betriebsmittel
- *I/O burst* inaktive Phase, nicht alle Betriebsmittel sind verfügbar (Betriebsmittel werden ausführlicher in *Kapitel 5.1 Betriebsmittel* abgearbeitet)
- Betriebsmittel werden durch andere Threads bereitgestellt.

3.1 Prozesszustände

Grobe Skizze:



- *long-term scheduling:*
  Lastkontrolle im Mehrprogrammbetrieb
- *medium-term scheduling:*
  swapping von *kompletten* Programmen und Adressräumen
- *short-term scheduling:*
  Einlastungsreihenfolge

Nur das short-term Scheduling ist obligatorisch, Spezialrechner brauchen die anderen unter Umständen nicht.

3.1.1 logische Zustände:

kurzfristig:

- bereit:
  Prozess ist auf *ready list* (Bereitliste), Listenposition abhängig von der Scheduling-Strategie
- laufend:
  Prozess läuft, zu einem Zeitpunkt nur ein Prozess pro CPU
- blockiert:
  Prozess wartet auf Betriebsmittel, zum Beispiel auf Beendigung eines I/O-Auftrags

mittelfristig:

- schwebend bereit:
  Prozessexemplar ist ausgelagert und wartet darauf eingelagert zu werden, die Einlastung aller Threads dieses Prozesses ist außer Kraft
- schwebend blockiert:
  Prozess ist ausgelagert und erwartet Ereignis

Schwebend bereite Prozesse werden i.d.R. neu erzeugten vorgezogen.

langfristig:

- erzeugt:
  Prozessexemplar wurde erzeugt, wartet gegebenfalls auf Speicherzuteilung
- gestoppt:
  Prozess wurde angehalten (Überlast, Verklemmmungsvermeidung ...).
  Laufende, bereite und blockierte Prozesse können gestoppt werden
- beendet:
  Prozess ist terminiert, Betriebsmittel wurden freigegeben, Prozess muss eventuell noch beseitigt werden (*zombie*)

(vgl. mit Folie IX.1/20)
Sobald ein Prozess in den Zustand bereit übergeht wird durch die Einplanungsfunktion die ready list aktualisiert.
Prozesse können dazu gedrängt werden die CPU abzugeben (preemtiv).
//irgendwas mit Einplanungs-/Einlastungslatenzen

3.2 Kriterien für Prozesseinplanung

- benutzerorientiert
  - Antwortzeit
  - Durchlaufzeit (Zeit vom Start bis zur Beendigung von einem Prozess minimieren)
  - Termineinhaltung
  - Vorhersagbarkeit
- Systemorientiert
  - Durchsatz
  - Prozessorauslastung
  - Gerechtigkeit
  - Dringlichkeit
  - Lastausgleich

Prozesseinplanung impliziert Rechnerbetrieb:

- allgemein:
  Gerechtigkeit, Lastausgleich

- Stapelbetrieb:
  Durchsatz, Durchlaufzeit, Prozessorauslastung
- Dialogbetrieb:
  Antwortzeit
- Echtzeitbetrieb:
  Dringlichkeit, Termineinhaltung, Vorhersagbarkeit

3.3 Einplanungsalgorithmen

3.3.1 Klassifikation

Kooperativ vs. Präemtiv

- **kooperative Planung:**
  - Ein-/Umplanung voneinander **abhängiger** Prozesse
  - CPU wird Prozessen nicht entzogen
  - CPU Monopolisierung ist möglich
  - Prozess kann CPU mittels Systemaufruf ab
- **präemtive Planung:**
  - Ein-/Umplanung voneinander **unabhängiger** Prozesse
  - CPU kann Prozessen entzogen werden
  - Prozess wird **ereignisbedingt** von der CPU verdrängt

Deterministisch vs. Probabilistisch

- **deterministische Planung:**
  - (fast) ***alles*** ist bekannt (à priori)
  - genaue Vorhersage ist möglich
  - Zeitgarantien sind sichergestellt
- **probabilistische Planung:**
  - es ist nicht alles bekannt
  - es sind nur Schätzungen möglich
  - keine Zeitgarantien

Abgrenzung ist eher fliesend.

Statisch vs. Dynamisch

- **vorlaufende Planung (*off-line scheduling*):**
  - **vor** Betrieb des Rechensystems
  - keine Planung im laufenden Betrieb
  - **vollständiger Ablaufplan**
  - meist beschränkt auf strikte Echtzeitsysteme
- **mitlaufende Planung (*on-line scheduling*):**
  - **während** Betrieb des Rechensystems
  - Stapelsysteme, interaktive Systeme, verteilte Systeme, schwache und feste Echtzeitsysteme

Abgrenzung auch hier nicht scharf (Ablaufpläne im Voraus zu haben ist supi).

Asymmetrisch vs. Symmetrisch

- **asymmetrische Planung:**
  - von Prozessoreigenschaften der Maschinenprogrammebene abhängig
  - jede CPU hat lokale Ready List
  - gegebenfalls ungleichmäßige Auslastung
  - keine gegenseitige Beeinflussung
  - obligatorisch in einem asymmetrischen Multiprozessorsystem (Spezialprozessoren)
- **symmetrische Planung :**
  - von Prozessoreigenschaften der Befehlssatzebene abhängig
  - keine Spezialprozessoren (alle gleich)
  - Prozesse gleichmäßig auf alle CPUs verteilen
  - globale Ready List

3.3.2 Planungsverfahren

FCFS (first come, fist served)

- FIFO-Liste, Prozesse kommen in der Reihenfolge ihrer Ankunftszeit dran
- gerecht
- im Mittel höhere Antwortzeit und niedrigerer I/O-Durchsatz
- Prozesse mit langen CPU-Bursts werden begünstig
- Voraussetzung sind kooperative Prozesse
- Konvoi-Effekt: kurze Prozesse folgen einem langen

RR (round robin)

- wie FCFS, nur werden Prozesse in periodischen Zeitabständen umgeplant
- periodische Programmunterbrechungen(mit Interrupts) und Zeitscheiben (*time slices*) (Obergrenze für Rechenstoßlänge)
- Zeitscheiben zu lang $\rightarrow$ FCFS; zu kurz $\rightarrow$ hoher Overhead
- Prozesse die kürzer rechnen als die Obergrenze werden trotzdem benachteiligt (I/O)
- I/O intensive Prozesse werden benachteiligt
- I/O Geräte werden schlecht ausgelastet
- Konvoi-Effekt

VRR (virtual round robin) RR mit zusätzlicher Vorzugswarteschlange und variablen Zeitscheiben

- in der Vorzugsliste werden Prozesse aufsteigend nach ihrem **Zeitscheibenrest** eingeplant
- Prozesse der Vorzugsliste werden zuerst eingelastet, für die verbleibende Dauer ihrer Zeitscheibe, danach werden sie der Ready List hinzugefügt

(vgl. Folie IX.2/20)

SPN (shortest process next)

- erwartete Bedienzeit wird abgeschätzt/ausgerechnet (Abschätzung entweder statisch oder dynamisch)
- aufsteigend nach Laufzeit sorierte Ready List
- Verkürzung von Antwortzeiten, Steigerung der Gesamtleisung und Benachteiligung längerer Prozesse (diese können *verhungern*)
- Abschätzung geschieht durch Mittelwertbildung der letzten Rechenstoßlängen und eventueller wichtung dieser

HRRN (highest response ratio next) Verhungerungsfreies SPN

- Prozesse werden nach ihrer erwarteten Bedienzeit eingeplant und periodisch unter Berücksichtigung ihrer Wartezeit umgeplant
- Je länger die Wartezeit eines Prozesses ist, desto höher die Priorität in der Ready List

SRTF (shortest remaining time first)

- Prozesse werden nach ihrer erwarteten Bedienzeit eingeplant und in unregelmäßigen Zeitabständen spontan umgeplant
- Falls die erwartete Rechenstoßlänge eines der Ready List hinzugefügten Prozesses kleiner ist, als die verbleibende Rechenstoßlänge des laufenden Prozesses, wird der laufende Prozess verdrängt
- die Umplanung ist ereignisbedingt (bei Aufhebung der Wartebedingung eines Prozesses) und verdrängend
- ein verdrängter Prozess wird anhand der Restdauer seiner erwarteten Rechenstoßlänge in die Ready List eingeordnet
- bessere Antwort-/Durchlaufzeiten
- mehr Aufwand gegenüber VRR (Schätzungen)

MLQ (multilevel queue)

- Einplanung der Prozesse nach ihrem Typ (zutreffend geglaubte Eigenschaft), dieser Typ ist fest
- Aufteilung der Ready List in separate Listen für die Typen (zum Beispiel für System-, Dialog- und Stapelprozesse)
- jede Liste kann eigene lokale Einplanungsstrategie haben
- zwischen Listen globale Einplanungsstrategie:
  - statisch:
    Liste hat feste Prioritätsebene, Verhungerungsgefahr für Prozesse in Listen mit niedriger Priorität
  - dynamisch:
    Listen Zeitmultiplexen (zum Beispiel 40% System-, 40% Dialog-, 20% Stapelprozesse)

FB (feedback) Begünstigung von kurzen Prozessen ohne Stoßlängen kennen zu müssen

- Hirachie von Ready Listen, neue Prozesse haben höchste Priorität
- jede Ebene der Hirachie hat verschiedene Einreihungsstrategien und Einreihungsparameter
- unterste Ebene RR, alle anderen FCFS
- Je niedriger die Priorität, desto größer die Zeitscheiben
- Prozesse mit langen Rechenstößen sinken in ihrer Priorität
- kurze Prozesse laufen schnell durch
- *anti-aging:* von alten Prozessen kann die Priorität angehoben werden

	FCFS	RR	VRR	SPN	HRRN	SRTF	FB
kooperativ	✓			(✓)	(✓)		
verdrängend		✓	✓			✓	✓
probabilistisch				✓	✓	✓	
deterministisch	keine/nicht alleine						

4 Arten von Threads

Da die Arten nicht explizit in der Vorlesung erwähnt wurden basiert Nachfolgendes auf Übung und das Glossar.

User-Threads ("federgewichtig")

- Anwendungsebene
- Kernel sieht nur einen Kontrollfluss, weshalb auf Multiprozessorsystemen kein Parallelismus möglich ist und alle User-Threads blockiert sind, sobald einer blockiert ist
- Erzeugung und Umschaltung sehr billig
- Programmierer kann über die Scheduling-Strategie der User-Threads entscheiden
- im lokalen Benutzer-Adressraum

leichtgewichtiger Prozess

- jeder Thread ist dem Betriebssystem bekannt
- Erzeugung und umschaltung erheblich teurer als bei User-Threads
- "Gemeinsam" mit anderen Prozessen seiner Art im gemeinsamen Adressraum

schwergewichtiger Prozess

- Prozessinkarnation in eigenem Adressraum
- teurer als leichtgewichtige-Prozesse

Leichtgewichtige und schwergewichtige Prozesse sind Kernel-Threads (Systemkernfaden).

5 Prozesse

Definition: Ein Programm in Ausführung. Ein Programm ist statisch, ein Prozess dynamisch. (Eine Prozessinkarnation steht zu einem Prozess wie ein Objekt zur Klasse)

5.1 Prozessinstanz

Definition: eine physische Instand des abstrakten Gebildes "Prozess"
eine konkrete Ausführungsumgebung
im Objektorientierten Sinn eine Instanz Durch Prozesse lassen sich

gleichzeitige Programmabläufe darstellen.

5.2 Betriebsmittel

konsumierbare Betriebsmittel

- unbegrenzt verfügbar
- werden erstellt, benutzt und sind danach weg
- **Harwarebetriebsmittel:** Signal, trap
- **Softwarebetriebsmittel:** Meldung, Datenstrom

wiederverwendbare Betriebsmittel

- begrenzt verfügbar
- werden angefordert, benutzt und danach wieder freigegeben
- sind entweder teilbar oder unteilbar
- **Harwarebetriebsmittel:** CPU, GPU, Speicher (RAM, harddrive?)
- **Softwarebetriebsmittel:** kritischer Abschnitt, Variable

6 Namen

- relativ eindeutige Bezeichnung von Dingen, die nach außen zugänglich ist
- relativ, da Namen nur in einem bestimmten Kontext eindeutig sind
- zum Beispiel symbolische Referenz auf Adresse (int x $\mapsto$ 0x.....)

6.1 Dynamische Namensauflösung (Dateisysteme)

- Adressen (names) die ein Programmierer benutzt (address space /name space)
- **kontextabhängig:** derselbe Name kann verschiedene Sachen adressieren
- dieselbe Adresse kann mehrere Namen haben
- interner Ort: Platz im Hauptspeicher
- externer Ort: irgendwo, aber nicht im Hauptspeicher
- Adresse als Standort ("Namen von Dateien im Ablagespeicher")

6.1.1 Hierarchischer Namensraum

- ein Kontext ist ein Namensraum flacher Struktur → Namen eindeutig
- in einem hierarchischen Namensraum können gleiche Namen in unterschiedlichen Kontexten auftreten (eindeutiger Pfadname)

Verzeichnisse

- ein **Verzeichnis** ordnet Namen listenförmig an, ist selbst im Elternverzeichnis (*parent directory*) gelistet, gibt mehreren Namen denselben Kontext
- **Unix:**
  - root directory: Einstiegspunkt in den Namensraum
  - home directory: initiales Verzeichnis eines *Users*
  - working directory: aktuelles Verzeichnis eines Prozesses
- **Informationsstruktur** (Indexknoten/Inode)
  - sammelt Attribute eines benannten Gegenstands (Eigentümer, Rechte, Timestamps, Typ, Anzahl Hard Links)
  - eindeutige numerische Adresse im Namensraum
  - Inode Number: nur im Namensraum eindeutig
- **Inode Table**:
  Array von Inodes

- **Verzeichnis**:
  Abbildungstabelle, übersetzt symbolische Namen in inode numbers
- **Datei**:
  abgeschlossene Einheit beliebiger Daten

Verzeichniseintrag

- Hard Link (Verknüpfung) von einem Dateinamen auf Inode Number
- mehrere Links auf eine Inode können existieren, im selben Verzeichnis jedoch mit unterschiedlichen Namen
- im Inode ist die Anzahl der Links auf ihn selbst gespeichert
- die Inode Number ist Eintrag in der Inode Table
- Anlegen/Löschen von Datein erfordert Schreibrechte auf das Verzeichnis, nicht nur auf die Datei
- **Namensverzeichnis:**
  - spezielle Datei die auch einen Namen hat, der inode bezeichnet
  - aus einem anderen Verzeichnis aus erreichbar
  - Namen werden getrennt von eventuellen Dateiinhalten gespeichert
- Verzeichnis hat mindestens die Einträge "." und ".." (im Root Directory sind beide eine Selbsreferenz)

(vgl. Folie 6.3/23)

Symblische Verknüfung (symbolic link)

- wegen hierarischer Struktur darf es keine Hard Links auf zum Beispiel das Root Directory geben
- Anders als Hard Links sind Symbolische Verknüfungen Dateien und besitzen einen Inode

6.2 Statische Namensauflösung

symbolische Adressen in Programmen müssen aufgelöst werden

1. Programmsymbole auf Binderabschnitte verteilen (Symbole werden bekannt gemacht und mit Attributen verknüpft, Adresswerte sind noch unbekannt)
2. Nach Assemblierung gibt es zwei Tabellen
   - **Symboltabelle:**
     assoziiert jedes Symbol aus den Modulen mit Werten
   - **Relokationstabelle:**
     listet alle unaufgelösten Referenzen
3. Text-/Datensegment besitzen vorläufige relative Adressen
4. undefinierte Adressen korrigiert der Binder/Lader

7 Speicherverwaltung: Zuteilungsverfahren

7.1 Verwaltung von Freispeicher

Ein freier Bereich im Hauptspeicher wird als Loch *(hole)* bezeichnet. Löcher haben feste oder variable Größe, weshalb Freispeicher auf verschiedene Arten dargestellt wird:

7.1.1 Bitkarte

- Stücke fester Größe, normalerweise Zweierpotenzen (Größe ist Vielfaches einer Seitengröße)
- für seitennummerierte Adressräume
- alle Löcher sind gleich gut
- jedes Stück hat ein Bit welches anzeigt ob das Stück einem Prozess gehört oder nicht
- Bitkarte wird als zweidimensionales Array gespeichert

7.1.2 Lochliste

- Stücke können verschieden groß sein, haben jedoch eine Mindestgröße (typischerweise Vielfaches der Größe eines Listenelements)
- dynamische Datenstruktur mit zwei grundlegenden Speicherausprägungen:
  - **Löcher in *anderem* Adressraum wie Liste:**
    Löcher sind wirklich leer und können beliebig klein sein. Listenelemente belegen Speicher im Adressraum des Betriebssytems, Listenoperationen wirken in der Schutzdomäne des Betriebssytems
  - **Löcher in *gleichem* Adressraum wie Liste:**
    Löcher sind scheinbar leer, sie beinhalten Listenelemente. Löcher haben Mindestgröße (mindestens so groß wie ein Listenelement). Listenoperationen wirken in einer anderen Domäne (Betriebssystem?).

Gebrauchsstück

- Speicher der von Prozessen für verschiedene Sachen gebraucht wird.
- gehört zum Betriebssystemadressraum, wird bei Zuteilung ausgeblendet und bei Zurücknahme wieder eingeblendet

7.1.3 Lineare Lochlisten

Sortierung nach Größe:

best-fit

- aufsteigende Lochgrößen, das kleinste passende Loch wird gesucht
- minimaler Verschnitt, jedoch langsam
- erzeugt vorne kleine Löcher, steigender Suchaufwand

worst-fit

- absteigende Lochgrößen
- sehr schnell, begünstigt Fragmentierung

- zerstört die großen Löcher, hinterlässt eher große Löcher, konstanter Suchaufwand

Bei beiden Verfahren müssen Restlöcher, die eine bestimmte Mindestgröße haben, wieder in die Liste einsortiert werden. Dafür muss ein zweites Mal durch die Liste iteriert werden.

Sortierung nach Adressen:

first-fit

- schnell, begünstigt aber Verschnitt
- erzeugt vorne kleine Löcher, steigender Suchaufwand

next-fit

- wie first-fit, die Suche beginnt jedoch beim zuletzt zugeteiltem Loch
- hinterlässt eher gleichgroße Löcher
- im Mittel eher abnehmender Suchaufwand

Bei beiden Verfahren fallen keine Restlöcher an.

Halbierungsverfahren (Buddy-Verfahren)

- Lochliste ist nach Zweierpotenzen aufsteigend sortiert
- bei einer Speicheranforderung wird zuerst der angeforderte Speicher auf die nächste Zweierpotenz aufgerundet
- Falls es keinen Block dieser Größe gibt, wird ein Block mit doppelter Größe gesucht, welcher dann in zwei Hälften (Buddies) geteilt wird. Ein Buddie wird dann dem Prozess zugewiesen
- Falls es keinen Block doppelter Größe gibt, wird nach einem Block vierfacher, achtfacher etc. Größe gesucht, welcher dann entsprechend oft geteilt wird
- schnell, jedoch kann viel Verschnitt entstehen
- Da die Buddies alligned werden unterscheiden sich die Adressen von zwei Buddies der Größe 2^n im Bit $n + 1$

7.2 Verschnitt

7.2.1 interne Fragmentierung

- Paging, Buddy-Verfahren
- angeforderter Speicher ist kleiner als das zugeteilte Stück
- Verschnitt ist nutzbar, sollte es aber nicht sein
- unvermeidbar

7.2.2 externe Fragmentierung

- segmentierte Adressräume, Buddy-Verfahren
- angeforderter Speicher ist zu groß für jedes einzelne Loch
- Verlust ist aufwändig vermeidbar

7.2.3 Verschmelzung und Kompaktifizierung bei externen Verschnitt

Verschmelzung

- Verschmelzung von Löchern erzeugt größere Löcher, was externer Fragmentierung entgegenwirkt und die Liste und damit den Suchaufwand verkürzt
- Falls ein Loch zwischen zwei Gebrauchsstücken ist, ist keine Verschmelzung möglich, sonst schon
- Der Aufwand hängt von der Art der Liste ab:
  - **buddy:**
    ein Stück kann nur mit seinem Buddy verschmolzen werden, Adressen von Buddies gleichen sich bis auf ein Bit
  - **first/next-fit:**
    beim Einsortieren Nachbarschaft prüfen
  - **best/worst-fit:**
    wie bei first/next-fit beim Einsortieren prüfen, Listennachfolger müssen jedoch keine Nachbarn sein, weshalb die Liste zweimal durchlaufen werden muss

Kompaktifizierung

- Stücke werden im Hauptspeicher verschoben, bis ein einzelnes großes Loch übrig bleibt
- Stücke müssen kopiert werden, entweder direkt im Hauptspeicher oder über Ablagespeicher
- Da sich die Lage der Stücke ändert müssen Adressen aktualisiert werden
- Anzahl der Umlagerungsvorgänge soll minimal sein

8 Prozesssynchronisation

Nebenläufigkeit: Aktionen die voneinander kausal unabhängig sind. Notwendige Bedingung für parallele Prozesse ist die Fähigkeit des Betriebssystems zur Simultanverarbeitung:

- **vertikal:**
  Multiplexen
- **horizontal:**
  mehrere Prozessoren

8.1 Koordination

- **off-line:**
  analytischer Ansatz, ermöglicht durch Vorabwissen. *Implizite* Synchronisation
- **on-line:**
  kein Vorabwissen, konstruktiver Ansatz. *Explizite* Synchronisation durch Programmanweisungen

8.2 Synchronisationsarten-/techniken

8.2.1 Wirkung

- **unterdrückend:**
  - verhindert Prozessauslösung anderer Prozesse
  - betrifft konsumierbare Betriebsmittel
  - aktueller Prozess wird nicht verzögert

- **blockierend:**
  - blockiert Betriebsmittelvergabe
  - betrifft wiederverwertbare/konsumierbare Betriebsmittel
  - aktueller Prozess kann suspendiert werden
- **nichtblockierend:**
  - betrifft Betriebsmittel *Speicher*
  - aktueller Prozess kann zurückgerollt werden

8.2.2 Einseitige Synchronisation

Betrifft lediglich einen beteiligten Prozess.

- **Bedingungssynchronisation:**
  - Fortschritt eines Prozesses ist von einer Bedingung abhängig
  - Der Prozess der diese Bedingung aufhebt wird nicht verzögert
- **logische Synchronisation:**
  - Maßnahme resultiert aus der logischen Abfolge von Aktivitäten?

8.2.3 Mehrseitige Synchronisation

Betrifft alle beteiligten Prozesse. Welcher Prozess fortschreitet ist unklar.

- **blockierend:**
  - pessimistisch, viel Konkurrenz
  - wechselseitiger Ausschluss gleichzeitiger Prozesse (multilateral)
  - profitiert von *Maschinenprogrammebene* (Systemaufrufe)
  - in der Regel zeitlich begrenzte Betriebsmittelvergabe
- **nichtblockierend:**
  - optimistisch, wenig Konkurrenz
  - basiert auf *Transaktionen* zwischen Prozessen (Folge von Aktionen die komplett oder gar nicht stattfnden)
  - spezielle Befehle der *Befehlssatzebene*
  - nicht geeignet für wiederverwendbare unteilbare Betriebsmittel (stand nicht oben schon, dass nichtblockierend nur für das Betriebsmittel Speicher funktioniert?)

8.2.4 Fortschrittsgarantien

Aussagen zur Lebendigkeit nichtsequentieller Programme, Merkmale nichtblockierender Synchronisation:

- **behinderungsfrei (*obstruction-free*):**
  - ein Isolierter Prozess wird seine Aktion in endlicher Anzahl von Schritten beenden
  - Ein Prozess ist isoliert, falls alle anderen Prozesse, die ihn blockieren können, nicht laufen
- **sperrfrei (*lock-free*):** (umfasst Behinderungsfreiheit)
  - jeder Schritt eines Prozesses lässt das nichtsequentiellen Programm fortschreiten
  - systemweiter Fortschritt garantiert, einige Prozesse können verhungern
- **wartefrei (*wait-free*):** (umfasst Sperrfreiheit)
  - die Anzahl der Schritte die für die Beendigung einer Aktion gebraucht werden hat eine obere Schranke
  - garantiert Systemweiten Fortschritt und ist frei von Aushungerung

8.3 Monitore

Erklärung: Ein Monitor ist ein abstrakter Datentyp, dessen Zugriffsoperationen implizit synchronisiert sind.

mehrseitige Synchronisation an der Monitorschnittstelle, mittels (vorzugsweise) Semaphor

einseitige Synchronisation innerhalb des Monitor mittels Bedingungsvariable (wait/signal)

- **Monitorprozeduren:**
  repräsentieren kritische Abschnitte: Prozeduraufruf sperrt Monitor, Prozedurrückkehr gibt Monitor frei
- **Monitorwarteschlange:**
  Prozessexemplare die Eintritt in den Monitor erwarten. Existiert nur wenn der Monitor durch einen anderen Prozess belegt ist
- **Ereigniswarteschlange:**
  Prozessexemplare im Monitor, welche die Aufhebung der Wartebedingung erwarten (das Ereignis, welches die Wartebedingung aufhebt ist konsumierbar)
- ein Prozess wartet stets außerhalb des Monitors, da sonst die Wartebedingung die aufgehoben werden könnte

8.3.1 Hansen, Hoare, Mesa und Java Monitore

Hansen:

- Monitorwarteschlange e mit Prozessen die den Monitor betreten wollen
- Signalnehmer werden nach e übertragen
- Vorzugswarteliste für Signalgeber, eventuell vereinigt mit e
- Je eine Ereigniswarteschlange pro Bedingungsvariable
- Signalgeber verlassen der Monitor, warten außerhalb. Wiedereintritt falls *signal* nicht letzte Operation
- blockierend: Signalnehmer hat Vorrang
- Signalisierung lässt den Signalgeber den Monitor verlassen, nachdem er alle Signalnehmer bereit gesetzt hat

Hoare:

- wie Hansen-Monitor, nur betreten Signalnehmer den Monitor direkt
- Signalisierung lässt den Signalgeber den Monitor verlassen und *genau* einen Signalnehmer fortfahren

Mesa:

- Monitorwarteschlange mit Zutrittsanforderern *und* mit den signalisierten Prozessen
- je eine Ereigiswarteschlange pro Bedingungsvariable
- Signalgeber fahren fort, "Sammelaufruf" möglich ($n > 1$ Signalisierungen)
- Signalnehmer starten erst nachdem der Monitor freigegeben wurde
- nichtblockierend: Signalgeber hat Vorrang
- Signalisierung lässt den Signalgeber im Monitor fortfahren, nachdem er einen oder alle Signalnehmer bereit gesetzt hat

Java

- Monitorwartschlange mit Zutrittsanforderern *und* mit den signalisierten Prozessen
- Ereigniswarteschlange für den gesamten Monitor
- Signalisierung wie bei Mesa

Zusammenfassung

	blockierend	einen	alle	Garantie	erneute Auswertung	Toleranz
Hansen	✓		✓		✓	✓
Hoare	✓	✓		✓		
Mesa		✓	✓		✓	✓
Java		✓	✓		✓	✓

Erklärung:

- deblockiert **einen/alle** wartenden Prozesse
- Wartebedingung ist **garantiert** aufgehoben
- eine **erneute Auswertung** der Wartebedingung ist nötig
- **Toleranz** gegenüber falschen Signalisierungen

8.3.2 Bedingungsvariablen

Variable ohne Wert. Fundamental für Bedingungssynchronisation. Zwei wichtige Operationen (druch den Compiler erzeugt):

- **signal:**
  - zeigt Ereignis an
  - wirkungslos, falls kein Prozess wartet
  - Führt dazu, dass genau *ein* wartender Prozess weiterläuft
- **wait:**
  - Prozess "schläft" bis zum Ereigniseintritt
  - gibt Monitor implizit frei

Ereignisvariable: Operationen cause/await. Cause befreit alle Prozesse die das gleiche Ereignis erwarten.

9 Semaphore und Sperren

9.1 Semaphore

Spezielle ganzzahlige Variable mit den Operationen P und V.

- **P:**
  - verringert den Wert des Semaphors um 1, wenn der Wert des Semaphors > 0 ist
  - ist der Wert des Semaphors 0 blockiert der Prozess
- **V:**
  - erhöht den Wert des Semaphors um 1
  - blockierte Semaphore werden aufgeweckt
- Die Operationen P und V sind physisch unteilbar.
- Man unterscheidet zwischen **binären** (Wert immer 0 oder 1) und **allgemeinen** (Wert hat obere und untere Schranke) Semaphoren
- Da P und V kritische Abschnitte sind müssen die Operationen abgesichert werden:
  - **pessimistischer Ansatz:** Annahme, dass gleichzeitige Operationen auf demselben Semaphor wahrscheinlich sind
    - **wechselseitiger Ausschluss**, der Semaphor hat eine Sperre mit der P und V geschützt werden
    - Blockierung eines Prozesses beim Aufruf von P muss den kritischen Abschnitt implizit Entsperren
    - Aufwecken von Prozessen mit V sollte nur erfolgen wenn Prozesse in P blockiert wurden
  - **optimistischer Ansatz:** Annahme, dass gleichzeitige Zugriffe unwahrscheinlich sind; benutzt nichtblockierende Synchronisation

(In den Folien stehen noch viele Sachen zur Implementierung und Codeschnipsel)

9.2 Mutex

Ein Semaphor kann von jedem Prozess freigegeben werden, ein Mutex jedoch nur von dem Prozess der ihn "besitzt".

Hinweis: "Prüfung der Berechtigung zur Freigabe eines kritischen Abschnitts ist ungeeignet für einen allgemeinen Semaphor, optional für einen binären Semaphor und notwendig für einen Mutex."

- **notwendig:**
  - Zusicherung, dass nur der Prozess den kritischen Abschnitt freigeben kann, der ihn erworben hat (**Mutex**)
  - Verwendung eines binären Semaphors mit Überprüfung des Besitzrechts
- **ungeeignet:**
  - Bei **allgemeinen Semaphoren** müssen die Operationen P und V für verschiedene Prozesse möglich sein
  - einseitige Synchronisation: Konsumenten und Produzenten (Bestimmte Prozesse warten auf ein Ereignis mit P, andere Prozesse signalisieren den Eintritt des Ereignisses mittels V)
- **optional:**
  - **binäre Semaphoren** lassen sich durch allgemeine implementieren
  - ungeeignet um kritische Abschnitte mit Prozesswechsel zu schützen

Bei unberechtigter Freigabe sollte der Prozess angehalten werden. Falls der Prozess im priviligierten Modus ist, sollte das Rechensystem angehalten werden. (Wie merkt man, dass eine Freigabe unberechtigt ist, wenn man schon nicht in der Lage ist die Berechtigung bei der Freigabe zu prüfen?)

spezieller binärer Semaphor:

- ein Mutex besteht aus zwei Komponenten:
  - einem binären Semaphor
  - etwas um Prozesse zu Identifizieren

- ein Mutex besitzt zwei Operationen:
  - **acquire:** vollzieht P und merkt sich den Prozess als Besitzer
  - **release:** Ausnahme, falls Prozess nicht der Besitzer ist, andernfalls wird der Besitzer zurückgesetzt und V aufgerufen

9.3 Sperre

Gleichzeitige Prozesse werden unterbunden, indem der Mechanismus für ihre Auslösung (dispatch?) deaktiviert wird. Interrputs deaktivieren? Damit werden aber auch Prozesse eingeschränkt die kausal unabhängig sind. Außerdem funktioniert dies nur prozessorlokal, ist also ungeeignet für Multiprozessorsysteme.
Second level interrupt handler blockieren die zur Auslösung des Schedulers führern.
Dabei wird nicht die Prozesseinplanung verzögert, sondern nur die Prozesseinlastung.
(Am besten nochmal in den Folien nachlesen)

10 Kreiseln und Spezialbefehle

10.1 Umlaufsperre

Dient zur Synchronisation von Prozessen auf verschiedenen Prozessoren. Prozesse durch "Kreiseln" steuern:

- **acquire:**
  wartet *aktiv* solange Sperre gesetzt ist, setzt die Sperre wenn sie nicht aktiv ist
- **release:**
  hebt Sperre auf

Bei naiven Implementierungen können viele Wettlaufsituationen auftreten. Acquire muss atomar erfolgen, was nur mit Spezialbefehlen der Befehlssatzebene machbar ist.

- **TAS (test and set):**
  - Operation schreibt immer in den Arbeitsspeicher, Werteveränderung jedoch nur bedingt

  - Bei Ausführung erfolgt ein wechselseitiger Ausschluss von Zugriffen auf den Speicherbus
  - atomarer Lese-/Schreibzyklus
  - in GCC eingebaute Funktion
  - Kreiseln (aktives Warten) mit TAS ist sehr schädlich, da der Bus immer gesperrt wird (interferiert auch mit kausal unabhängigen Prozessen) und der Cache stark beansprucht wird (Cache muss oft invalidiert und geupdated werden)

- **CAS (compare and swap):**
  - **Signatur:** CAS(Variable, Prüfwert, Neuwert)
  - CAS vergleicht den Wert der Variable mit dem Prüfwert, falls die Werte gleich sind wird der Variable der neue Wert zugewiesen
  - Cache wird nicht so stark belastet wie bei TAS, Interferenzen mit kausal unabhängigen Prozessen bestehen trotzdem, da Buszugriffe synchronisiert werden müssen

- **Kreiseln mit ablesen:**
  - vor einem CAS-Aufruf wird mit einer Schleife (while(lock→busy)) gewartet bis das lock frei ist, um die Anzahl von CAS-Aufrufen zu senken
  - es entstehen keine Buszugriffe, da in der Schleife nur auf den Cache zugegriffen wird (muss der Cache in Multiprozessorsystemen dann nicht aktualisiert werden was wiederum zu Interferenzen führt?)
  - bei viel Konkurrenz stauen sich viele Prozesse (Schleife) und der Bus wird häufig gesperrt

- **Kreiseln mit Zurückhaltung: (backoff)**
  - Nach jedem unerfolgreichen Aufruf von CAS wartet der Prozess eine vorgegebene Zeit, bis er erneut probiert die "kritische Aktion" auszuführen

- **Verhungern (starvation):**
  - Obige Verfahren sichern gekoppelten Prozessen Fortschritt zu, jedoch können einzelne Prozesse ewig kreiseln

  - deshalb wird versucht eine obere Schranke für Wartezeiten zu garantieren

- **Kreiseln mit proportionaler Zurückhaltung: (*ticket spin lock*)**
  - Wettstreiter werden gezählt
  - es wird ein Prozess ausgewählt, welcher als nächster die kritische Aktion ausführen darf
  - Prozesse welche nicht an der Reihe sind die Aktion auszuführen warten eine bestimmte Zeit

Zusammenfassung Umlaufsperren

- mehrseitig
- verdrängende Prozesseinplanung verträgt sich nicht mit der Benutzung von Umlaufsperren (CPU kann dem Schlosshalter entzogen werden etc.)
- Interferenzen mit anderen Prozessen und Verklemmungsgefahr

10.2 Transaktion

Definition Transaktion (Glossar) "Aktion oder Aktionsfolge, die eine logische Einheit bildet und als Ganzes entweder gelingt oder scheitert."

Nachteile blockierender Softwaresynchronisation

- **Leistung:**
  Kreiseln reduziert Busbandbreie, Parallelität wird eingeschränkt
- **Robustheit:**
  Fehler im kritischen Abschnitt führen eventuell zum Systemabsturz
- **Interferenz:**
  Entscheidungen von Scheduler werden beeinflusst
  (Prioritätsverletzungen etc.)
- **Lebendigkeit:**
  Gefahr von Verhungerung und Verklemmung

Kreiseln mit Bedingung:

- nichtatomare Aktionsfolge
- Datum wird lokal berechnet und dann durch speziellen Hardwarebefehl getauscht, falls die notwendige Bedingung passt
- Dadurch wird die Wartezeit nicht nutzlos verschwendet, sondern für Rechnungen genutzt

11 Stillstand

11.1 Verwaltung von Betriebsmitteln

Buchführung über die vorhandenen Betriebsmittel und Steuerung von Betriebsmittelanforderungen sind zu bewältigende Aufgaben.
Falls ein Prozess sich nicht ordnungsgemäß verhält können diesem Betriebsmittel entzogen werden.

Verfahrensweisen

- **statisch**
  - vor Laufzeit
  - Anforderungen davor
  - Zuteilung erfolgt vor der eigentlichen Benutzung
  - Freigabe *aller* Betriebsmittel mit Laufzeitende

  ↪ eventuell keine optimale Auslastung

- **dynamisch**
  - zur Laufzeit
  - Anforderung bei Bedarf
  - Zuteilung "im Moment" der Benutzung
  - Freigabe, falls Prozess mit Betriebsmittel fertig ist

  ↪ Risiko von Verklemmung

11.2 Systemblockade

Definition deadly Embrace: Gekoppelte Prozesse warten auf die Aufhebung einer Wartebedingung, welche nur durch einen der gekoppelten, wartenden Prozesse aufgehoben werden kann.

- **inaktives Warten**
  - Prozess schläft (Befehlszähler konstant)
  - CPU wird abgegeben
  - **"gutartig"**, da es erkannt werden kann
- **aktives Warten**
  - Prozess läuft (Befehlszähler ändert sich)/ist *träge*, wechselt zwischen laufend und bereit
  - **"bösartig"**, da es nicht erkannt werden kann

11.3 Voraussetzungen für Verklemmungen

notwendige Bedingungen

- wechselseitiger Ausschluss
- Nachforderung von Betriebsmitteln (die der Prozess momentan nicht besitzt, aber braucht)
- Betriebsmittel können nicht entzogen werden

notwendige und hinreichende Bedingung

- zirkuläres Warten (in PFP wurde damit gemeint, dass alle benötigten Betriebsmittel sequentiell und nicht atomar angefordert wurden)

Falls eine dieser Bedingungen nicht erfüllt ist kann keine Verklemmung entstehen.

11.4 Verklemmungsvorbeugung (*deadlock prevention*)

indirekte Methoden

- nichtblockierende Synchronisation

- *alle* benötigten Betriebsmittel atomar anfordern
- Betriebsmittel virtualisieren: reale Betriebsmittel können entzogen werden, die Prozesse behalten jedoch die virtuellen

direkte Methoden

- Betriebsmittel ordnen; Zuteilung erfolgt nur in richtiger Reihenfolge

11.5 Verklemmungsvermeidung (*deadlock avoidance*)

- notwendige Bedingungen werden nicht aufgehoben
- zirkuläres Warten wird durch laufende Analyse verhindert
- vor jeder Betriebsmittelanforderung wird überprüft, ob das System in einen unsicheren Zustand kommen kann

Betriebsmittelgraph

- Stellt Prozessexemplare und Betriebsmittel (eventuell nach Klassen geordnet) zusammenhängend dar
- wird mit Vorabwissen angelegt und mit Fortschreiten der Prozesse aktualisiert
- Durch Analyse kann Zyklenbildung erkannt werden

Bankiersalgorithmus (*bankers algorithm*)

- Es wird angenommen, dass das System weiß welche Betriebsmittel jeder Prozess möglicherweise anfordert, welche jeder Prozess besitzt und welche verfügbar sind
- der Zustand des Systems ist sicher, falls ein Prozess seinen möglichen Bedarf oder die Verfügbarkeit einer Betriebsmittelklasse, bei einer Anfrage nicht überschreitet

11.6 Verklemmungsnachweis und Erholung

- Es erfolgt kein Versuch Verklemmungen vorab zu vermeiden
- Es wird mittels eines Wartegraphen nach blockierten Prozessen gesucht
- Danach werden Verklemmungen entfernt:
  - das Prozessexemplar, welches die Verklemmung auslöst wird zerstört, gegenfalls werden alle beteiligten Exemplare zerstört
  - andernfalls wird ein Prozess ausgesucht, welcher zurückgerollt wird und dem die Betriebsmittel entzogen werden

Verfahren zum Vermeiden/Erkennen von Verklemmungen sind eher *praxisirrelevant*, da sie aufwendig und schwer umzusetzen sind.

12 Dateisysteme

12.1 Festplatten

Aufbau

- Mehrere einzelne Platten mit je einem Plattenarm mit Schreib-/Lesekopf
- Einzelne Platten sind in Spuren aufgeteilt, Spuren sind in Sektoren aufgeteilt
- Sektoren von Spuren unterschiedlicher Platten, die übereinander sind, werden Zylinder genannt
- Sektoren bestehen aus einer Präambel und Datenbits
- Mehrere Zylinden werden zu Zonen zusammengefasst, Zonen haben die gleiche Anzahl von Sektoren
- Ein Abschnitt fester Größe wird Block genannt, zwischen 32 und 4096 Bytes groß (normalerweise 512 Bytes)
- Blöcke wurden früher durch Zylindernummer und Sektornummer eindeutig identifiziert, heutzutage haben Blöcke eine eindeutige Nummer

12.2 Speicherung von Dateien

12.2.1 Kontinuierliche Speicherung

- Dateien werden in aufeinanderfolgenden Blöcken gespeichert
- Dazu wird die Nummer des ersten Blocks und die Anzahl der Blöcke gespeichert
- **Vorteile:**
  - minimale Positionierungszeit des Plattenarms
  - schneller Zugriff auf bestimmte Dateiposition
  - unter anderem eingesetzt in Echtzeitsystemen
- **Nachteile:**
  - Finden von aufeinanderfolgenden freien Blöcken ist aufwendig

  - Fragmentierung (Platz zwischen zwei Dateien ist zu klein um etwas abzuspeichern)
  - Falls Dateien größer werden muss eventuell die ganze Datei umkopiert werden

12.2.2 Verkettete Speicherung

- Block einer Datei hat immer einen Zeiger auf den nächsten Block
- **Vorteile:**
  - Datei kann einfach vergrößert werden
- **Nachteile:**
  - Speicher für Zeiger geht verloren (nicht gut im Zusammenhang mit Paging)
  - Fehleranfälligkeit (falls ein Zeiger falsch ist ist möglicherweise die ganze Datei weg)
  - schlechter Zugriff auf bestimmte Dateiposition
  - häufiges Umpositionieren vom Lese-/Schreibkopf
- **FAT-Ansatz** (*File Allocation Table*):
  - Zeiger werden nicht in den Blöcken mit Daten gespeichert, sondern in einem speziellen Block
  - **Vorteile:**
    * innerhalb der Blöcke geht kein Speicher verloren (gut für Paging)
    * weniger Fehleranfällig da der FAT mehrfach gespeichert werden kann
  - **Nachteile:**
    * Zusätzlich zu den Blöcken mit Daten muss die FAT geladen werden
    * Laden von FAT-Blöcken lädt auch Informationen über andere Dateien, welche nicht benötigt werden
    * Suche nach Datenblöcken bei bekannter Dateiposition ist aufwendig
    * häufiges Umpositionieren vom Lese-/Schreibkopf

12.2.3 Indiziertes Speichern

- Jede Datei hat einen Indexblock fester Größe in dem zur Datei gehörende Blöcke gespeichert werden
- **Problem ist die feste Größe der Indexblöcke:**
  - Verschnitt bei kleinen Dateien
  - Bei großen Dateien ist eine Erweiterung nötig

mehrstufige Indizierung

- die ersten n Verweise eines Indexblock verweisen direkt auf Datenblöcke
- die nächsten k verweisen wiederum auf Indexblöcke etc.
- **Vorteile:**
  - Da Inodes kleiner Dateien sowieso einen Block auf der Platte brauchen ist Verschnitt unproblematisch
  - große Dateien sind adressierbar
- **Nachteile:**
  - für die Adressierung von großen Dateien müssen eventuell mehrere Blöcke geladen werden

12.3 Freispeicherverwaltung

- ähnlich wie Freispeichverwaltung von Hauptspeicher
- Belegung von Blöcken durch Bitvektoren
- Listen für freie Blöcke (Blöcke können zusammengefasst werden/freier Block enthält Nummern weiterer freier Blöcke)

12.4 Beispiel: UNIX Dateisyteme (Blockorganisation)

- Es gibt einen **Boot Block** mit Informationen die zum Systemstart benötigt werden
- Ein **Super Block** enthält Informationen zu den Inodes (Anzahl von belgten/freien Blöcken/Inodes, Attribute)

BSD 4.2 (Berkley Fast File System)

- Ist in Zylindergruppen aufgeteilt
- Jede Zylindergruppe hat einen Super Block
- freie Inodes/Blöcke im **Cylinder Group Block**
- Dateien werden wenn möglich innerhalb einer Zylindergruppe gespeichert
- BSD 4.2 hat den Vorteil von kürzeren Positionierungszeiten

Linux EXT2/3/4

- Anstelle der Aufteilung in Zylindergruppen wird in Blockgruppen aufgeteilt
- **EXT3:** Erweiterung als Journaling-File-System
- **EXT4:** Extents

12.5 Extents

- Zusammenhängender Speicherbereich wird als Spanne (Range) gespeichert, also mit den zwei Werten *von* und *bis*
- Eine Datei besteht aus mindestens einem Extent
- **Vorteile:**
  - Speicherung von zweie Nummern anstatt aller Blocknummern
  - weniger Fragmentierung
  - Weniger Metadaten

(Quelle: Wikipedia)

12.6 Beispiel: Windows NTFS (New Technology File System)

- **Datei:**
  - beliebiger Inhalt
  - Dateien können automatisch komprimiert/verschlüsselt werden
  - Dateien bis zu 2^{64} Bytes groß
- **Dateiinhalt:**
  - Sammlung von *Streams*: Folge von Bytes
  - "normaler Inhalt" = unbenannter Stream (*default stream*)
  - dynamisch erweiterbar
- **Cluster:**
  - Größe: 512 Bytes bis 4 Kibibytes (Festgelegt bei Formatierung)
  - Menge hintereinanderfolgender Blöcke
  - *Logical Cluster Number* als Adresse
- **Strom (*Stream*):**
  - jede Datei kann mehrere Streams speichern
  - ein Stream für Dateiinhalt

  - Name, Rechte, Attribute etc. werden in separaten Streams gespeichert

- **File-Reference:**
  - 64-Bit Zahl
  - Niederwertigen 48 Bits (Dateinummer): Index in Master-File-Table
  - Höherwertigen 16 Bits (Sequenznummer): wird für jede Datei mit gleicher Dateinummer erhöht

12.6.1 Master-File-Table (MFT)

- Tabelle mit gleich großen Einträgen (abhängig von Clustergröße)
- dynamisch erweiterbar
- entsprechender Eintrag einer File-Reference enthält Informationen über die Streams der Datei
- **Einträge die immer in der MFT sind:**
  - Standart-Information: Länge, Standart-Attribute, Zeitstempel, Anzahl der Hard-Links, Sequenznummer
  - Dateiname
  - Rechte (?)
- bei *kurzen* Dateien werden die Daten im Eintrag der MFT gespeichert
- bei *längeren* Dateien wird die Virtual Cluster Number, LCN und Anzahl der Cluster in der MFT gespeichert, Extents werden außerhalb gespeichert
- bei einem *Katalog* werden die File-References, Dateinamen und Längen der Dateien im Katalog direkt in der MFT gespeichert (es wird auch eine Belegungstabelle gespeichert)
  Große Kataloge werden mithilfe von Extents gespeichert (Als B^+ Baum)
- Metadaten zur MFT werden in Dateien gehalten: MFT, MFT-Kopie, Log File, Attributtabelle, Boot File etc.

12.7 Journaling-File-System

Änderungen von Meta-Daten werden protokolliert.

- Änderungen sind Transaktionen (werden entweder vollständig ausgeführt oder zurückgerollt)
- beim Bootvorgang werden Inkonsistenzen mit der Protokolldatei beseitigt
- Jeder Einzelvorgang einer Transaktion wird *vor* der eigentlichen Änderung protokolliert
- Beim Bootvorgang können Transaktionen abgeschlossen (redo) oder zurückgerollt werden (undo)
- Da die Log-File nicht beliebig wachsen kann werden an bestimmten Punkten (Checkpoint) für einen konsistenten Zustand gesorgt und dieser protokolliert. Vorherige Protokolleinträge können gelöscht werden)

12.8 Copy-on-Write-/Log-Structured-File-Systems

- Konzept für atomare Änderungen
- Änderungen erfolgen auf Kopien, danach können die Zeiger von den alten Blöcken auf die neuen zeigen
- Alte Daten können behalten werden (einfache snapshots)
- **Vorteile:**
  - Gute Schreibeffizienz (wenn immer komplette Kopien gemacht werden?)
  - Datenkonsistenz bei Systemausfällen

- **Nachteile:**
  - Starke Fragmentierung, kann Lesezeiten erhöhen
- Log-Structured-File-Systems schreiben ans Ende des belegten Bereichs und geben vorne Blöcke frei

12.9 Datensicherung mit RAID

RAID 0

- Daten werden über mehrere Platten gespeichert
- Datentransfers schneller
- Ausfall einer Platte zerstört alle Dateien

RAID 1

- Daten werden auf mehreren Platten dupliziert gespeichert
- schnelles Lesen, n-facher Speicherbedarf (bei n Platten)
- Schreiben eventuell langsamer

RAID 4

- mindestens drei Platten
- letzte Platte enthält Paritätsinformationen (XOR)
- eine Platte kann ausfallen, schnelles Lesen, Paritätsplatte wird am stärksten beansprucht

RAID 5

- wie RAID 4 nur werden die Paritätsinformationen auf alle Platten verteilt

RAID 6

- Zwei Paritätsblöcke
- bis zu zwei Platten können ausfallen

13 Programme

13.1 Übersetzten und Binden

1. **Präprozessor**
   entfernt Kommentare, wertet Präprozessoranweisungen aus
2. **Compiler:**
   C → Assembler
3. **Assembler:**
   Assembler → Objekt-Datei (Maschinencode, Informationen über static Variablen, Symboltabelle (globale Variablen/Funktionen), Relokierungsinformation ("nicht gefundene" Referenzen))
4. **Binder:**
   erzeugt ausführbare Datei (ELF-Format), Objekt-Dateien werden zusammengebunden (unaufgelöste Referenzen in andere Objekt-Dateien werden aufgelöst (*Relokation*))
   - **statisch binden:**
     - fehlende Funktionen werden in die ausführbare Datei **kopiert**.
   - **dynamisch binden:**
     - unaufgelöste Referenzen werden nicht aufgelöst (diese liegen in einer *shared library*)
     - Relokation erfolgt beim Laden

13.2 Speicherorganisation

definiert durch das ELF-Format

13.3 Laden von Programmen

statisch gebundene Programme:

- Teile der ausführbare Datei wird in den Speicher geladen
- Speicher wird bereitgestellt
- argc und argv werden in den Speicher kopiert, ihre Zeiger initialisiert
- main-Funktion wird angesprungen

dynamisch gebundene Programme:

- spezielles Lade-Programm: **ld.so**
- Programm wird in den Speicher geladen
- Speicher wird bereitgestellt
- fehlende Funktionen werden aus den shared libraries geladen
- Relokation
- argc/argv/main-Funktion wird angesprungen (wie bei statisch)
- weitere Funktionen können zur Laufzeit nachgeladen werden (plugins)

14 Betriebsarten

14.1 Stapelbetrieb

Verfahren um bereitgestellte Aufträge automatisch nacheinander abzuarbeiten. Eingabedaten werden dabei im Voraus bereitgestellt, Ausgabedaten werden gespeichert oder ausgegeben. Die Aufträge sind in einem Stapel hinterlegt und werden ohne externe Befehle abgearbeitet.

14.2 Dialogbetrieb/Mehrzugangsbetrieb

- Abwechslung von Benutzereingabe und Verarbeitung
- on-line scheduling
- mehrere Programme laufen gleichzeitig ab
- Mischbetrieb realisierbar (z.B. Vordergrund Echtzeitbetrieb, Mittelgrund: Dialogbetrieb, Hintergrund: Stapelbetrieb)
- Ziel des Dialogbetriebs ist es Antwortzeiten *und* die Dauer bis zur Fertigstellung der Aufträge zu minimieren

14.3 Echtzeitbetrieb

Das Rechensystem muss dafür sorgen, dass Prozesse beim Verarbeiten von Daten zu einem bestimmten Zeitpunkt (*deadline*) fertig sind.

Merke: Geschwindigkeit ist keine Garantie, auch ist die Laufzeit von Prozessen irrelevant, solange sie termingerecht fertig sind.

Terminvorgaben:

- **weich (*soft*):**
  Terminverletzungen tolerierbar, Ergebnisse können trotz Terminverletzung von Nutzen sein
- **fest (*firm*):**
  Terminverletzungen tolerierbar, Ergebnisse werden jedoch wertlos (der Prozess wird abgebrochen)
- **hart (*hard*):**
  Terminverletzung *keinesfalls* tolerierbar, **Ausnahmesituation**

15 Traps und Interrupts

15.1 Trap/Synchrone Ausnahme

- interne Ursache
- z.B. ungültige Adresse, unbekannter Befehl, Teilen durch "0", Systemaufrufe, Seitenfehler (bei lokaler Ersetzungsstrategier, bei globaler → Interrupt)
- vorhersagbar, reproduzierbar
- Behandlung im Fehlerfall zwingend

15.2 Interrupt/Asynchrone Ausnahme

- externe Ursache
- z.B. Beendigung einer I/O-Operation, Seitenfehler bei globaler Ersetzungsstrategie
- unvorhersehbar, nicht reproduzierbar
- *Nebeneffektfreiheit* der Behandlung ist *zwingend*