Betriebssysteme II

Zusammenfassung v1.0

Kälin Thomas, Abteilung I
SS 06

1. Sicherheit (627)

1.1. Begriffe

- Schutzstrategie / Policy (630)
  - Was ist zu schützen? → Festlegung der Schutzobjekte
  - Wie soll der Schutz wirken? → Festlegung der Schutzziele
  - Ist anforderungs-, bzw. bedarfsorientiert
- Schutzmechanismus / Mechanism (630)
  - Wie wird geschützt? → Legt Realisierung des Zugriffschutzes fest
  - Ist umsetzungs-, bzw. technikorientiert
- Hochsichere Betriebssysteme (Trusted Solaris, SELinux)
  - Prinzip minimaler Privilegien, Benutzer / Prozess soll zu jedem Zeitpunkt nur zwingend nötige Rechte besitzen
  - Preis für Mehrsicherheit ist hoch!

1.2. Schutzziele / Gefährdungen (627 / 628)

Ziel	Gefährdung
Vertraulichkeit: Kein Ausspionieren	Abfangen (Interception)
Integrität: Kein unerlaubtes Verändern	Modifikation (Modification)
Verfügbarkeit: Daten immer verfügbar	Unterbrechung (Interruption)
Authentisierung: Benutzer identifizierbar	Fabrikation (Fabrication)

1.3. Berechtigungsprüfung (628)

Rechtezuordnung	Rechteüberprüfung
Personalisierung Benutzer werden unterschieden (Benutzernamen)	*Identifizierung* Benutzer werden eindeutig identifiziert (PW)
Autorisierung Benutzer werden Rechte zugeordnet	*Zugriffskontrolle* Prüfung der Autorisierung beim Zugriff

1.4. Schutzsystem

- Benutzerverwaltung: Realisiert Personalisierung (Wer ist wer?)
- Anmeldesystem: Realisiert Identifizierung (Ist er es eindeutig?)
- Autorisierung und Zugriffskontrolle: Kernfunktion (Darf er Zugriff ausführen)
  - Nachfolgend beschränken wir unsere Betrachtung auf diese Punkte

1.5. Schutzdomänenkonzept (631)

- *Schutzdomäne (protection domain):* Menge von Objekten mit bestimmten Rechten
  - Pro Schutzdomäne eine Liste mit Objekt/Recht-Paaren
  - Objekte sind beispielsweise Dateien, Prozesse, Datenbanken, Semaphoren, ...
  - Falls Domäne = Benutzer -> Standardsicherheit, Pro Benutzer eine Domäne, Domänenwechsel bei An-/Abmelden
  - Falls Domäne = Prozess -> Hochsicherheit, Pro Prozess eine eigene Domäne, Domänenwechsel bei Prozesszwechsel
- *Recht (right)*: Erlaubnis, bestimmte Operationen auf Objekte auszuführen

1.6. Schutzmatrix (632-637)

1.6.1. Beispiel einer Schutzmatrix

	F1	F2	P1	P2	D1	D2
D1	R*	R-W-X	*owner* W	W	switch	*control* switch
D2		R		W		switch

- **switch (Domänenwechselrecht)**: Die Domänenwechsel dürfen nur kontrolliert erfolgen. Lesebeispiel: D1 darf nach D1 oder D2 wechseln. D2 darf nur in der eigenen Domäne verbleiben.
- *** (Kopierrecht)**: Recht, einen Eintrag zu kopieren. Lesebeispiel: D1 darf das Leserecht für F1 (R*) nach D2 kopieren.
  - *Copy*: Volles Kopierrecht (R*)
  - *Transfer*: Ein Recht wird kopiert, jedoch am Ursprungsort gelöscht

  - *Limited Copy*: Limitiertes Kopierrecht, * wird nicht mitkopiert (R)
    - Windows: In der Regel handelt es sich um ein limitiertes Kopierrecht. Es gibt jedoch die Möglichkeit für Copy und Transfer. Dazu müssen die Rechte „Berechtigung lesen“ und „Berechtigung ändern“ gesetzt sein.
- **owner (Besitzerrecht)**: Eignerkonzept sagt aus, dass jedes Objekt genau einen Besitzer hat. Alle Besitzer haben folgende Verwaltungsrechte: Rechte einfügen, Rechte entfernen, Besitzerrechte weitergeben. Lesebeispiel: D1 ist Besitzer von P1. D1 darf allen anderen Domänen die Recht auf P1 geben / nehmen.
  - Variante: Ein Objekt hat mehrere Besitzer, sehr selten
  - Verwaltungsrechte: create / delete object, create / delete domain, insert / remove right
  - Der Eigner hat IMMER die Verwaltungsrechte. Dadurch wird vermieden, dass keine Objekte im System existieren, auf die niemand mehr Verwaltungsrechte besitzt.
- **control (Kontrollrecht)**: Verwaltungsrecht zum Ändern aller Rechte einer Domäne. Lesebeispiel: D1 hat das Kontrollrecht auf D2. Es kann alle Rechte von D2 auf allen Objekten verändern!
  - Ein eigentliches Kontrollrecht existiert in Windows nicht.
- **Erweiterung (allow / deny)**: Hierzu müssten aus jedem Recht zwei Rechte gemacht werden
  - Read+: Lesen ALLOW
  - Read-: Lesen DENY

1.6.2. Eigenschaften / Verwendung der Schutzmatrix

- Stellt modellhaften Schutzmechanismus dar
- Illustriert Möglichkeiten der dynamischen Rechteverwaltung
- Modellierung und Beurteilung praktischer Implementierungen

1.6.3. Implementierung der Schutzmatrix (637)

- Komplette Matrix zu speichern ist ineffizient, da Matrix schwach belegt, wir brauchen Alternativen
- **Globale Tabelle (637)**
  - Liste mit sortierten Tripeln der Art <domain-id, object-id, rights-set>
  - (D1,F1,R*), (D1,F2,R-W-X), ...
- **Zugriffskontrollliste / ACL (637 / 639)**
  - Pro Objekt eine Liste von <domain-id, rights-set>
  - F1: -> (D1,R*) | F2: -> (D1,R-W-X), (D2,R) | ...
  - (+) Sind Objekten zugeordnet, damit diese bei einer Zugangskontrolle schnell gefunden werden
  - (+) Bei Objekterzeugung können gleichzeitig Berechtigungen gesetzt werden
  - (-) Schwierig herauszufinden, auf welche Objekte eine Domäne welche Rechte hat.
  - (-) Schwierig, die Rechte eines gelöschten Benutzers sauber zu entfernen
  - (-) Zugriffsprüfung muss für jeden Zugriff wiederholt werden
  - *Verwendung: Windows und Unix (vereinfachte Form)*
- **Ticket-Liste / C-List (638 / 639)**
  - Pro Schutzdomäne eine Liste von <object-id, rights-set>
  - D1: -> (F1,R*), (F2,R-W-X), (F3, owner-W), ...
  - (+) Rechte pro Domäne sind zusammengefasst gespeichert.
  - (+) Rechte können beim Löschen eines Benutzers einfach entfernt werden.
  - (-) Langes Absuchen der Liste bei Zugriffsprüfung.
  - (-) Technische Realisierung anspruchsvoll

1.7. Schutzstrategien (639)

- Benutzerbestimmte Zugriffskontrolle, Discretionary Access Control (DAC)
  - Objekte kennen Besitzer, dieser hat volle Verwaltungsrechte
  - Windows / Unix
- Systembestimmte Zugriffskontrolle, Mandatory Access Control (MAC)
  - Rechte sind an Rollen gekoppelt, organisatorische Regeln schränken die Rechteverwaltung ein
  - Hochsichere Betriebssysteme (Trusted Solaris, SELinux)

1.7.1. Bell/La Padula (640)

- Idee: Geheimnisse bewahren, Dokumente besitzen Sicherheitsstufen
  - *Simple Security Rule*: Leseberechtigung gilt für gleiche und alle tieferen Stufen (General kann Dokumente eines Leutnants lesen)
  - **-Rule*: Schreibberechtigung gilt nur für gleiche oder höhere Stufe (General darf keine Nachricht an Leutnant senden, da er geheimere Dokumente kennt.)

1.7.2. Biba-Modell (640)

- Idee: Datenintegrität gewährleisten
  - *Simple Integrity Rule*: Schreibberechtigung gilt für gleiche und alle tieferen Stufen (Leutnant kann seine Daten und die aller ihm Unterstellten ändern)
  - *Integrity *-Rule*: Leseberechtigung gilt nur für gleiche und höhere Stufe (Leutnant kann seine und die des Generals lesen)

1.8. Sicherheit unter Unix (642-645)

- Zentrales Unix-Konzept: Dateien & Prozesse
- Datei kennt einen Besitzer (UID) und eine Gruppenzugehörigkeit (GID)
- Jede Datei besitzt Rechteinformationen für owner, group und other
- Befehle: chmod (Rechte ändern), chown (Besitzer ändern), chgrp (Gruppe ändern)
- Dateien:
  - R -> Lesen (4),
  - W -> Schreiben / Löschen (2)
  - X -> Ausführen (1)
- Ordner:
  - R -> Verzeichnis lesbar mit ls (4)
  - W -> Einträge erstellen, löschen (2)
  - X -> Verzeichnis durchsuchbar (1)
- Spezialrechte:
  - S -> Setuid, Ausführung d. Datei mit Rechten d. Besitzers anstatt d. aufrufenden Benutzers (4)
  - G -> Setgid, Ausführung d. Datei mit Rechten d. Dateigruppe anstatt d. aufrufenden Gruppe (2)
  - T -> Sticky, Löschen nur durch Besitzer oder root erlaubt (1)

```
#: chown tom test.txt    //Neuer Besitzer von test.txt ist tom
#: chmod 0777 test.txt   //Vollzugriff auf test.txt
#: chmod 0600 test.txt   //Lesen/Schreiben für Besitzer auf test.txt
#: chmod 7777 test.txt   //Vollzugriff auf test, alle Spezialrechte gesetzt
#: chmod 5777 test.txt   //Sticky-Bit & SetUid-Bit gesetzt, Vollzugriff
```

1.8.1. Pseudobenutzer

- Problem: Ein Spiel speichert die Highscores in der Datei „highscore“. Es soll nun verhindert werden, dass die Systembenutzer manuell Einträge in dieser Datei erstellen / ändern können.
  - Wir erstellen einen speziellen Benutzer für das Spiel. Dieser ist Besitzer der ausführbaren Spieldatei. Zusätzlich setzen wir das SETUID-Bit (siehe oben) auf die Datei. Dadurch wird die Spieldatei immer mit den Rechten des Besitzers ausgeführt. Auf die Datei „highscore“ werden nur Lese-/Schreibrechte für diesen speziellen User gewährt.

1.9. Sicherheit unter Windows (Übung 2)

- Domänen: Benutzer, Benutzergruppen.
- Identifizierung: Benutzer werden inter über eine ID (SID = Security Identifier) angesprochen.
- Eignerkonzept: Jede Datei hat mindestens eine Owner-ID (SID) und ACL. Der Owner der Datei hat immer Verwaltungsrechte auf die Datei. Siehe auch 1.6.1 Stichwort „owner“.

2. Speicherverwaltung (401)

2.1. Einordnung im Rechnermodell (402)

- Primärspeicher: Hauptspeicher, Arbeitsspeicher (RAM / ROM) → Baublock „Main Memory“
  - Direkt Adressierbar
  - Physische Datenorganisation (nummerierte Speicherplätze > Speicheradressen)
  - Quelle für Instruktionen und Operanden
  - Flüchtiger Speicher
- Sekundärspeicher: CD, Harddisk → Baublock „Input / Output“
  - Indirekt Adressierbar (mittels Schnittstellen-Hardware und Software)
  - Logische Datenorganisation (Speicherverwaltung, Dateisystem)
  - Persistente Programm- und Datenspeicherung
- MMU (Memory Management Unit): Erlaubt die Implementierung eines virtuellen Speichersystems. Wird beispielsweise bei der Prozessumschaltung gebraucht.
- Pentium: Zugriff auf Input/Output erfolgt über eigenen Ein-/Ausgabe-Adressraum
- PowerPC: *Memory Mapped I/O*. Peripheriebausteine sind im Hauptsiecheradressraum sichtbar und werden darüber angesprochen.

2.2. Grundlegende Speicherprinzipien (403)

- Direktadressierbarer Speicher (403)
  - Adresse wird direkt über die gewünschte Speicherstelle angesprochen.
  - Beispiel: Hauptspeicher
- Mehrport-Speicher (404)
  - Lese-/Schreibspeicher mit mehreren Zugriffspfaden → Problem der Datenkonsistenz!
  - Gleichzeitiger Zugriff von mehreren Verarbeitungselementen auf gleichen Speicherbereich
- Schieberegister-Speicher (405)
  - Kette von ein Bit grossen Speicherzellen, „Fliessband-Prinzip“
  - Anwendung: Umwandlung Seriell – Parallel / Parallel – Seriell
  - Beispiele: USB / SATA / Ethernet
- FIFO-Speicher (405)
  - Arbeitet nach dem „Senioritätsprinzip“. Es wird immer dasjenige Datenelement ausgelesen, dass am längsten im Speicher ist.
  - Beispiel: Tastatur, Drucker, serielle Schnittstelle
- Stapel-Speicher (406)
  - LIFO-Speicher. Der zuletzt eingespeicherte Wert wird als erster wieder ausgelesen.
  - Anwendung: Als Stack mittels direktadressiertem Speicher und Stapelzeiger
- Assoziativ-Speicher - CAM: Content Adressable Memory (406)
  - Teil-Information eines Eintrages führt zu ganzem Informationseintrag
  - Für Zugriff ist keine Adresse notwendig, sondern ein Muster, über welches eine Prüfmaske gelegt wird
  - Anwendungen: Datenbankabfragen, Memory-Cache-Zugriff
  - Probleme: Mehrfachtreffer möglich, Verwaltung von freien Speicherstellen

2.3. Speicherhierarchie (407)

- Ziele für Speichersysteme sind: Minimale Zugriffszeit, Minimale Kosten pro gespeichertem Bit
- Der Awender, bzw. das Anwenderprogramm sieht einen einzigen, grossen Adressraum, ist sich der Hierarchie also nicht bewusst (Transparenz).
- Mehrstufiger Speicher
  - Register > SRAM (Cache) > DRAM (Hauptspeicher) > Massenspeicher
  - Prozessornahe Stufen sind schnell, teuer, kleine Kapazität. Prozessorfern -> umgekehrt!
- Funktionsprinzip:
  - Daten, auf die der Prozessor zugreift, müssen in CPU-nahe Stufe transferiert werden. Schneller Zugriff ist nur gewährleistet, wenn die Daten in der schnellsten Stufe sind
- Realisierung:
  - Hardware koordiniert prozessornahe Speicherstufen (Cache / Hauptspeicher)
  - Betriebsystem koordiniert prozessorferne Stufen (Hauptspeicher / Massenspeicher)

2.4.Cache (411)

2.4.1. Allgemeines / Grundprinzip (411 / 413)

- Der Cache kann in den Leveln 1-3 vorkommen. Dieser ist entweder prozessorintern (auf dem Chip, am schnellsten) oder prozessorextern. Er liegt somit zwischen der CPU und dem Hauptspeicher.
- Zweck: Schnellstmöglicher Speicherzugriff für Prozessor, Hauptspeicherzugriff verlangsamt für gewöhnlich den Prozessor
- Grundprinzip: Kleiner, schneller Cache-Speicher puffert Teilbereiche des grossen Hauptspeichers. Dieser Speicher enthält häufig gebrauchte Ausschnitte aus dem Hauptspeicher. Zur Identifizierung muss die Hauptspeicher-Adresse zusammen mit den Speicherinformationen im Cache enthalten sein.

2.4.2. Lokalitätseffekt (409)

- Beim Lokalitätseffekt wird die Tatsache ausgenutzt, dass ein Problem bei seiner Ausführung für gewöhnlich einen nahe zusammen liegenden Speicherbereich belegt. Dies ist der sog. Arbeitsbereich. Beim Laden von Daten des Hauptspeichers in den Cache wird dieser Effekt ausgenützt, indem benachbarte Speicherblöcke gleichzeitig in den Cache kopiert werden. Ausserdem kann davon ausgegangen werden, dass benötigte Datenblöcke erneut aufgerufen werden (Schleifen).

2.4.3. Grundstruktur des Cache-Speichers

- Eine Zeile besteht aus: Gültigkeitsbit (1-bit), Blocknummern und den Cache-Zeilen zu je 2^i Byte

```
Cache-Speicherkapazität = Anzahl Zeilen * Anzahl Byte pro Cache-Zeile
Cache-Speicherkapazität = 2^k * 2^i
Grösse des Tag (Bit) = log2(Adressraumgrösse / Cache-Zeilengrösse)
Grösse des Tag (Bit) = log2(2^m / 2^i) = log2(2^(m-1)) = m-1
```

2.4.4. Cache-Leistung (413, 416)

```
Teff = h * Tc + (1-h) * Tm
Teff: Effektive Zugriffszeit
Tc = Cache-Zugriffszeit, Tm: Speicher-Zugriffszeit
h: Trefferrate (Cachehit) beim Speicherzugriff (Praxis 0.9 .. 1.0)
```

- Trefferrate (h) wird beeinflusst von: Grösse des Cache-Speichers, Grösse einer Cache-Zeile, Organisationsform, Ersetzungsstrategie, Programmstruktur

2.4.5. Cache-Anwendungen (412)

- Hardware
  - Pufferspeicher zwischen CPU / Hauptspeicher, Pufferspeicher in MMU, Lokaler Pufferspeicher bei Peripheriegeräten (Festplatte, Netzwerkschnittstelle)
- Software
  - Pufferspeicher des Betriebssystems für Plattenspeicher, Pufferspeicher für Datenbanken

2.4.6. Ablauf eines Lesevorgangs (415)

- CPU verlangt 1 Byte an Lesedaten auf Adresse X
- Cache-Logik: extrahiere Blocknummer aus Zugriffsadresse X
- Cache-Logik: Prüft „tag-info“in allen Cache-Zeilen mit Gültigkeitsbit = 1
- TREFFER: „tag-info“ == Blocknummer, Lies1 Byte aus Zeile
  - Zugriff beendet
- NIETE: Keine Übereinstimmung im Cache gefunden
  - Kopie aller Bytes des Blocks von Hauptspeicher in Cache laden, „tag-info“ eintragen und Gültigkeitsbit auf 1 setzen. Byte aus Zeile lesen und an CPU liefern.
  - Hinweis: Falls Cache voll ist, muss eine Ersetzungsstrategie entscheiden, welche Zeile überschrieben wird. Für gewöhnlich LRU (Least Recently Used).

2.4.7. Cache-Kohärenz (415)

- Dieser Effekt tritt auf, wenn zwei verschiedene Kontrolleinheiten (bsp: Multiprozessorsysteme, intelligente Controller) auf einen gemeinsamen Cache zugreifen.
- Problem: Es muss die Datenkonsistenz gewährleistet werden.
- Lösung: Beispielsweise Cache-Kohärenzprotokolle. Logisches System, dass die Cache-Zugriffe überwacht und synchronisiert, bzw. blockiert.

2.4.8. Probleme des Cache-Speichers (415)

- Zugriffsadresse des Prozessors ist nicht im vornherein bekannt -> Lokalitätseffekt. Daraus folgt, dass Treffer im Cache erst beim zweiten Zugriff auf dieselbe Adresse erfolgen (beim ersten wird aus RAM geladen)
- Bei vollem Cache: Bei jedem Miss muss eine Cache-Zeile überschrieben werden

2.5. Verwaltungsgrundlagen des Speichers (416)

2.5.1. Grundbegriffe(420)

- Namensraum..
  - eines Programms (a, x, main): Menge aller Namen, die ein Programm zur Identifizierung von Elementen enthält. Wird auch Symbolmenge genannt.
  - eines Computersystems: Menge aller technisch möglichen Namen (Syntaxregeln)
- Adressraum...
  - eines Programms: Menge aller Programmadressen, die ein Programm bei seiner Ausführung referenziert
  - eines Computersystems: Menge aller technisch möglichen Programmadressen. Begrenzt durch Anzahl Adressbits -> 32bit = 2^3 = 4GB
- Speicherraum...
  - eines Programms: Menge aller Speicheradressen, auf die ein Programm bei seiner Ausführung zugreift
  - eines Computersystems: Menge aller Speicheradressen, die mit Speicherbausteinen hinterlegt sind. Entspricht konkreter Speicherbestückung (tatsächlich zugreifbare Speicherstellen).

2.5.2. Abbildungsfunktionen (417)

```
Quellcode Namen    -- Namensauflösung --> Programmadressen (CPU)
Programmadressen   -- Adressumsetzung --> Speicheradressen (Zugriff)
```

- **Namensauflösung**: Dem Namen in einem Quellcode wird eine Programmadresse zugeordnet
  - Bindezeitpunkte: Je später, umso flexibler. Ein gebräuchlicher Compiler bindet bei der Programmübersetzung. Ein Stackframe hingegen erst bei der Programmausführung.
    - Je später der Bindezeitpunkt (Abbildung von Namen auf Adresesn), desto grösser der Aufwand, jedoch umso flexibler der Programmbetrieb.
  - Kann entfallen, falls Quellcode keine Namen / Symbole enthält. In Praxis kaum realisierbar.

2.5.3. Adressumsetzung (417 / 459)

- Programmadressen (virtuelle Adressen) werden Speicheradressen (physische Adressen) zugeordnet.
- Virtuell sind die Adressen zusammenhängend, real jedoch diskontinuierliche Speichernutzung
- Realisierung der Umsetzung durch MMU (falls vorhanden) zusammen mit dem Betriebssystem. Bei Mikrokontrollersystemen verzichtet man meist auf Adressumsetzung.
- Dank der Adressumsetzung kann ein Mehrprozess-System realisiert werden (Pro Prozess ist je ein eigener, virtueller Adressraum verfügbar. Bedingt eine Umsetzungstabelle pro Prozess!).
- Durchführung der Adressumsetzung
  - MMU halt für jeden Speicherzugriff die Umsetzungsinformationen aus der Umsetzungstabelle. Die Adresse der Tabelle wird im **Zeigerregister** der MMU abgelegt.
  - Bei jedem Prozesswechsel wird das Zeigerregister neu geschrieben.
  - Da bei jedem Speicherzugriff zuerst die Umsetzungstabelle gelesen werden muss, besitzt die MMU einen **Translation Lookaside Buffer** (TLB), der als Cache-Speicher für die Tabelle fungiert. Es wird ein Assoziativspeicher (CAM-Speicher) eingesetzt.

2.5.4. Varianten den Speichersystems (421 / 422)

- A: Realer Speicher, Monoprogrammierung: Keine Adressumsetzung. Mikrokontrollersys., MS-DOS.
- B: Realer Speicher, Multiprogrammierung: Keine Adressumsetzung. Frühere Unix-Systeme.
- C: Virtueller Speicher, Multiprogrammierung: Adressumsetzung. Linux, Windows, Aktuelle Unix.

direkt (1:1)	$A_P = A_L$	Real	A,B
basisversetzt	$A_P = A_L+B$	Real	B
virtuell (seitenbasiert)	$A_P = A_L+v*S$	Virtuell	C
virtuell (segmentbasiert)	$A_P = A_L+A_S$	Virtuell	C
virtuell (segment- /seitenb.)	$A_P = A_L+A_S+v*S$	Virtuell	C Windows

A_P: Speicheradr. A_L: Programmadr. B: Basiswert S:Seitengrösse

2.6. Dynamische Speicherbereitstellung (423)

- Dynamische Daten haben eine Lebensdauer, die kleiner als diejenige des Programms ist. Reservierung und Freigabe erfolgen während des Ablaufs.
- Beispiele: temporäre Variablen, Funktionsparameter, lokale Variablen, Meldungspuffer, Objekte
- Realisierung:
  - Funktionsbezogene Lebensdauer -> Stack (lokale Variablen, Funktionsparameter, ...)
  - Applikationsdefinierte Lebensdauer -> Heap (Meldungspuffer, Objekte, ...)

2.6.1. Anforderungen (425)

- Flexible Zuordnungsgrösse, Zusammenhängende Zuordnungsbereiche, Schnellstmögliche Zuordnung, Maximale Speichernutzung, Adressraumausrichtung.
- Die Anforderungen sind leider NICHT widerspruchsfrei.

2.6.2. Speicherfragmentierung (425 / 426)

- **Externe Fragmentierung**: Durch unkoordinierte Reservierung und Freigaben mit unterschiedlichen Anforderungsgrössen kann eine Speicherzerstückelung entstehen. Die Folge sind nicht nutzbare Restbereiche (Verschnitt).
- **Interne Fragmentierung**: Intern bereitgestellte Bereiche passen nicht zu Anforderungsgrössen. Es folgt Verschnitt durch inflexible Speicherverwaltung. Ein Bereich ist intern nicht voll ausgefüllt.
- Problem: Reservierungsanforderung kann nicht befriedigt werden, da nicht genügend zusammenhängende Bereiche vorhanden sind.
  - Lösung: **Kompaktierung / Speicherverdichtung (427)**. Defragmentierung des Heaps durch Verschieben belegter Elemente. Dabei müssen auch die Startadressen der Blöcke geändert werden. In der Praxis wird dies mittels Master Pointern gelöst (Referenz A > MP A > Speicherbereich A).
  - Rekombination: Verbinden benachbarter Freibereiche (jeweils bei Bereichsfreigabe)

2.6.3. Heap: Verwaltungsdaten

- Verwaltungsdaten können dezentral, innerhalb der belegten Bereiche, oder zentral, innerhalb eines vordefinierten Bereiches, angelegt werden.
  - Bei der ersten Lösung wird der Speicherplatz nur dann belegt, wenn er tatsächlich auch gebraucht wird. Bei der zweiten Methode muss der Platz vorreserviert werden.
  - Bei einem Überlauf werden in der dezentralen Variante Daten zerstört, da der Zeiger des neuen Bereichs überschrieben wird. Bei der zentralen Ablage ist der Schutz gewährleistet.

2.6.4. Heap: C-Funktionen (429)

```
int * ptr = (int *) malloc(4 * sizeof(int));     //Speicher anfordern
ptr[3] = 42;                                     //Zugriff über Index
free(ptr);                                       //Speicher freigeben
```

- Anmerkung: Heap-Verwaltung weiss aus Verwaltungsdaten, wie viel Speicher das freizugeben ist.

2.6.5. Heap: Interne Organisation

- **Variante A: Variable Zuordnungsgrösse (431)**
  - Reservationsgrösse = Anforderungsgrösse. Hierdurch kann keine interne Fragmentierung entstehen.
  - Die Heap-Verwaltung führt Buch über freie Bereiche (verkettete Liste aus Startadresse, Grösse und Verkettungszeiger).
  - Es kann keine konstante Reservierungszeit garantiert werden, da die Suche nach einer passenden Lücke unterschiedlich lange dauern kann.
  - Suchalgorithmen
    - First Fit: Es wird die erste passende Lücke gewählt, fängt immer am Anfang des Speichers an. Hierdurch entsteht eine Massierung kleiner unnützer Lücken am Speicherbeginn.
    - Next Fit: Wie „First Fit“, allerdings ist Suchbeginn an der Stelle der letzten reservierten Lücke. Verteilung der Speicherlücken auf den ganzen Speicher.
    - Best Fit: Durchsucht die Liste nach der am besten passenden Lücke. Es entstehen viele kleine, unnütze Lücken. Ausserdem langsam.
    - Worst Fit: Durchsucht die Liste nach der grössten verfügbaren Lücke.
- **Variante B: Feste Blockgrössen / Grössenklassen (433)**

- Es gibt intern mehrere verschiedene Listen mit vorgegebenen Blockgrössen. Eine Anforderung wird dabei automatisch auf die nächste passende Grösse aufgerundet. Es kann somit interne Fragmentierung entstehen.
- Suchalgorithmus ist Quick Fit: Getrennte Listen für die Grössen. Da jede freie Lücke gleichwertig ist, kann stets die erste freie Lücke gewählt werden.
- Listenausgleich: Bei erschöpfter Liste einer Grösse, wird die nächste höhere Stufe in zwei kleinere Bereiche aufgeteilt. Umgekehrt können zwei tiefere Stufen zu einer verbunden werden. Diese Funktionalität ist jedoch Implementierungsabhängig und muss nicht immer vorhanden sein!

- **Variante C: Mehrfaches einer festen Blockgrösse (434)**
  - Es existiert genau eine Blockgrösse. Eine Anforderung wird mit der aufgerundeten, passenden Anzahl an Blöcken befriedigt. Hierdurch kann interne und externe Fragmentierung entstehen.
  - Verwaltung mittels Bitliste (Bitreihenfolge: frei=0, belegt=1) oder über verkettete Liste (Zustand, Startadresse, Grösse, Verkettungszeiger).

```
Platzbedarf Bitliste
 Speichergrösse: 256MB, Blockgrösse 1MB
 -> 256MB / 1MB = 256 Bit
Platzbedarf verkettete Liste
 Speichergrösse 256MB, Blockgrössen 1MB, Speicher pro Knoten: 16Byte
 -> 256 / 1MB = 256 Bereiche
 -> 256 Bereiche * 16Byte = 4kB
```

- **Variante D: Buddy-System (436)**
  - Flexibles Verfahren zur optimalen Bereitstellung freier Blöcke. Ein anfänglicher Block wird solange zweigeteilt, bis passende Blockgrösse erreicht. Speicherausnutzung liegt zwischen 50..100%. Prinzipbedingt kann interne und externe Fragmentierung entstehen.
  - Der erste Block befriedigt immer die Anforderung, der zweite Block wird in Freiliste seiner Grössenklasse eingetragen.
  - Bei der Rekombination können nur Buddys zusammengefasst werden. Zwei Blöcke sind Buddys, wenn Sie sich nur an einer Bitposition unterscheiden.

```
4 Byte Blockgrösse = 2^k => k=2 -> Bitposition 2 darf ungleich sein.
Beispiel: 0000 und 0100 sind Buddys!
```

2.7. Verwaltung von Prozessadressräumen (443)

Das Betriebssystem muss über die Belegungen des Adressraums Buch führen. Nur so ist es in der Lage, neue Reservierungen ohne Kollisionen durchzuführen.

2.7.1. Adressraumnutzung durch Programme (443)

Bei der Programmübersetzung gruppieren die Übersetzungswerkzeuge die einzelnen Programmteile gemäss der Adressraumnutzung in benannte Sektionen.

2.7.2. Inhalt Unix-Programms (444)

- **Kennung:** Steht ganz am Anfang, meistens eine sog. „magische Zahl“.
- **Programmstartadresse**
- **Sektionsbeschreibungen:** Startadressen, Sektionstypen und Sektionsgrössen
- **.text:** Enthält Maschinencode des Programms (Programm-Code)
- **.data:** Enthält Werte für Variableninitialisierung

2.7.3. Betriebssystemplatzierung im Adressraum (445)

Applikationen und Betriebssystem teilen sich den Adressraum.

2.7.4. Regionen, Adressraumverwaltung (446 / 447)

- Im Adressraum belegte Bereiche werden Regionen genannt. Typische Attribute einer Region sind: Startadresse, Grösse, Schutzattribute (benötigte spezielle Hardwareunterstützung) und der zugehörige Hintergrundspeicher.
- Die Verwaltung der Bereiche wird für gewöhnlich als verkettete Liste geführt.
- Windows verwendet für jeden Prozess einen ausgeglichenen Binärbaum (VAD’s)

2.8. Realer Speicher (448)

2.8.1. Monoprogrammierung (449)

- Es wird immer nur 1 Programm gleichzeitig ausgeführt (Bsp.: MS-DOS, Embedded Systems).

- Betriebssystem und Benutzerprogramm teilen sich vorhandenen Speicher, die Programmgrösse wird durch den verfügbaren physischen Speicher begrenzt.

2.8.2. Multiprogrammierung mit festen Partitionen (449-452)

- Es können mehrere Programme echt parallel oder quasiparallel ausgeführt werden
- **Speicheraufteilung:** Einfachste Möglichkeit besteht darin, dass der vorhandene Speicher in N feste Bereiche (=Partitionen) aufgeteilt wird, welche sich in der Grösse jedoch unterscheiden können. Ablaufbereite Prozesse werden auf diese Partitionen verteilt. Sind die Bereiche erschöpft, werden Warteschlangen geführt.
  - Gemeinsame Warteschlange für alle Bereiche, eventuell schlechte Speicherauslastung (kleiner Prozess in grosser Partition!)
  - Verteilte Warteschlange: Gute Speichernutzung, aber eventuell lange Warteschlangen bei bestimmten Partitionen.
- **Relokation:** Umrechnen der Adresse entsprechend der Partitionsstartadresse
- **Speicherschutz:** Damit Programme nicht ungehindert in fremde Partitionen schreiben können, wird jedem Prozess ein eigener Zugriffsschlüssel vergeben. Benötigt allerdings spezielle Hardware zur Umsetzung!
- **Basis Register:** Teil der CPU, löst das Problem der Relokation und des Speicherschutzes. Bei jedem Speicherzugriff wird automatisch das Basis-Register zur Adresse addiert. Die Werte des Basis-Registers müssen für jeden Prozess geführt werden (PCB!).

2.8.3. Modellierung der CPU-Ausnutzung (453)

$A = 1 - p^n$	A = Auslastung der CPU p = Zeitanteil für das E/A-Warten n = Anzahl Prozesse im Speicher

2.8.4. Verfahren für knappen Speicher (454-457)

- **Overlay-Technik:** Eignet sich gut für Monoprogrammierung. Basiert darauf, dass nur die momentan benötigten Programmteile in den Hauptspeicher geladen werden.
- **Swapping:** Prozesse werden auf die Festplatte ausgelagert, wenn Sie keine Rechenzeit benötigen. Echtzeit-Prozesse müssen im Speicher fixiert werden!

2.9. Virtueller Speicher (457)

- Virtueller Speicher: Jeder Prozess hat scheinbar den ganzen Adressraum für sich. Dadurch hat auch jeder Prozess seinen privaten Adressraum, ist somit geschützt gegen Fehlzugriffe.
- Um virtuellen Speicher realisieren zu können muss eine Adressumsetzung (Abbildung von logischen auf physische Adressen) verwendet werden. Mehr dazu unter „Verwaltungsgrundlagen des Speichers“.

2.9.1. Segmentbasierte Adressumsetzung (461)

$$A_P = A_L + A_S$$

- Teile eines Prozesses werden auf Segmente (= zusammenhängender Adressbereich beliebiger Grösse) aufgeteilt. Jedes dieser Segmente trägt eine eindeutige Identifikation.
- Um die physische Adresse (A_P) zu ermitteln, wird jedem Segment (A_L) ein bestimmter Verschiebungsfaktor (A_S) zugewiesen.
- Die Umsetzungstabelle enthält alle A_S-Werte der vorhandenen Segmente, zusätzlich die jeweilige Segmentlänge. Pro Segment ist natürlich ein Eintrag vorhanden.
- Als Speicherschutz wird jedem Segment eine Limite (= Segmentlänge) zugeordnet, die beim Zugriff überprüft wird.
- Programmadresse (Logisch) sind Tupel bestehend aus Segmentnummer und Relativadresse. Die Segmentnummer dient als Index in die Umsetzungstabelle, aus welcher die physische Segmentstartadresse gelesen wird. Zu dieser wird anschliessend die Relativadresse addiert, wodurch die physische Adresse komplettiert wird.
- Vorteile: Passgenaue Bereiche (kein Verschnitt), Geringer Platzbedarf für Umsetzungstabellen
- Nachteile: Externe Fragmentierung, Hauptspeicherbewirtschaftung aufwendig

2.9.2. Seitenbasierte Adressumsetzung (463)

$A_P = P_S * S + A_R$	$A_R = A_L \% S$	$P_V = A_L / S$
A_P = Physische Adresse	A_R = Relative Adresse	A_L = Logische Adresse
S = Seitengrösse	P_V = Vir. Seitennummer	P_P = Phy. Seitennummer

- Teile eines Prozesses werden auf Bereiche fester grösser (= Seiten) aufgeteilt. Im Hauptspeicher (physisch) wird dieselbe Aufteilung vorgenommen (= Seitenrahmen).
- Für gewöhnlich wird als Blockgrösse eine Zweierpotenz gewählt. Dadurch lassen sich einige weitere Berechnungen sehr einfach umsetzen:

```
Seitengrösse       = 2^n Byte    = 2^3 Byte    => n=3
Log. Adresse       = 0x1870F     = 1000111
Seitennummer       = 1000                      => Letzte 3 Bits abschneiden
Relative Adresse   = 111                       => Letzte 3 Bits relevant

Seitengrösse       = 2^Länge der Relativadresse          = 2^3        = 8 Byte
Anzahl Seiten      = 2^Länge der Seitennummer            = 2^4        = 16 Seiten
Speichergrösse     = Seiten * Seitengrösse               = 16*8Byte   = 128 Byte
```

- Wie oben ersichtlich, besteht eine virtuelle Adresse aus einer Seitennummer und einer Relativadresse. Die Seitennummer dient als Index für die Umsetzungstabelle. Aus dieser wird die Physische Rahmennummer gelesen, welche mit der Seitengrösse multipliziert wird. Zu diesem Produkt wird anschliessend noch die Relativadresse addiert, wodurch die physische Adresse komplett ist.
- Die Umsetzungstabelle enthält einen Einträg für jede Seite. Zusätzlich wird pro Eintrag (Deskriptor) ein „Valid"-Bit und ein „Write/Read"-Bit verwaltet.
- Der Platzbedarf für die Umsetzungstabelle errechnet sich aus der Seitengrösse multipliziert mit der Grösse eines Seitendeskriptors. Hieraus ist sofort ersichtlich, dass bei kleinerer Seitengrösse der Verwaltungsaufwand ansteigt. Umgekehrt steigt bei grösserer Seitengrösse der Verschnitt innerhalb der Seiten. (Siehe auch Seite 468)
- Vorteile: Einfache Hauptspeicherverwaltung, keine externe Fragmentierung
- Nachteile: Interne Fragmentierung wahrscheinlich, Grosser Platzbedarf für Umsetzungstabellen

2.9.3. Seitenwechselverfahren – Demand Paging (471)

- Es wird sich der Lokalitätseffekt zu Nutze gemacht, indem nur der aktuelle Arbeitsraum in den Speicher geladen wird. Der Rest des Programms wird auf der auf einem Hintergrundspeicher (Harddisk) seitenweise ausgelagert und bei Bedarf in den Hauptspeicher geholt.
- Der Unterschied zum Swapping ist, dass beim Seitenwechselverfahren nur Teile des Codes ausgelagert werden. Beim Swapping hingegen ein kompletter Prozess.
- Vorteile: Geringer Ladeumfang, weniger Hauptspeicherplatz benötigt, kürzere Reaktionszeiten
- Nachteil: Zusätzlicher Platzbedarf auf HD, evt. Spürbare Nachladezeiten
- Die effektive, durchschnittliche Zugriffszeit (476) errechnet sich wie folgt:

```
T_eff    = (1-p)*T_M + p*T_PF
T_eff    = Zugriffszeit (Mittel)
T_M      = Zugriffszeit Hauptspeicher
T_PF     = Zeitdauer des Seitenwechsels (~Plattenzugriffszeit)
p        = Wahrscheinlichkeit eines Seitenfehlers
```

- Wegen des Lokalitätseffektes kann die Häufigkeit der Seitenwechsel stark reduziert werden. Bei nicht vorhandenem Lokalitätseffekt nimmt die Seitenfehlerrate linear mit den prozentual geladenen Seiten eines Prozesses ab (478).
- **Dreschen (479):** Kann bei zu knappem Hauptspeicher und vielen Prozessen auftreten. Da die MMU sehr viele Seitenwechsel durchführen muss, wird die CPU nur noch sehr schlecht ausgelastet (Wartezeit!).
  - Als Lösung für dieses Problem eignet sich eine Laststeuerung. Aber einer gewissen Anzahl Prozesse dürfen keine neuen Prozesse mehr gestartet werden.
  - Eine Alternative ist die Laststeuerung aufgrund der Seitenfehlerrate (480). Hierbei wird bei knappem Hauptspeicher abhängig von der Seitenfehlerrate die Platzzuteilung eines Prozesses verändert. Steig beispielsweise die Fehlerrate über einen definierten Wert, wird der zugewiesene Speicherplatz (geladene Seiten) vergrössert.
- **Verdrängungstrategien (484):** Bei Hauptspeicherknappheit müssen bereits belegte Seitenrahmen mit einer der folgenden Strategien neu zugeteilt werden.
  - FIFO-Algorithmus: Entfernt die Seite, die am längsten im Speicher war
  - LFU-Algorithmus: Entfernt die Seite, welche am wenigsten verwendet wurde
  - LRU-Algorithmus: Entfernt die Seite, die am längsten nicht verwendet wurde. Beste Variante.
  - OPT-Strategie: Geht davon aus, dass wir wissen, welche Rahmen in der Zukunft gebraucht werden. In der Praxis natürlich unmöglich, deshalb nur theoretischer Natur.

  - Clock-Algorithmus: Variante des LRU. Technisch einfacher zu realisieren und liefert eine gute Annäherung an den echten LRU. Genaue Funktionsweise siehe Buch Seite 486.
- **Ladestrategie (483):** Entscheidet, wann Seitenrahmen zugeordnet werden (Ladezeitpunkt).
  - Zuteilung nach Bedarf: Bei Seitenfehler wird nur der betreffende Block geladen
  - Zuteilung im Vorraus: Bei Seitenfehler wird der betreffende und benachbarte Blöcke geladen
- **Entladestrategie (490):** Entscheidet, wann modifizierte Speicherinhalte auf Platte geschrieben werden
  - Entladen nach Bedarf: Nur beim Ersetzen der Seite
  - Entladen im Voraus: frühzeitiges schreiben der Seite, beschleunigt spätere Ersetzung
  - Seitenpufferung: Es werden gemäss einer Verdrängungstrategie Seiten ausgewählt. Diese werden in eine von zwei Listen (modified / unmodified) eingeordnet. Die Seiten aus modified werden in regelmässigen Abständen auf die Harddisk geschrieben und nach unmodified verschoben. Bei einem Seitenfehler wird nun aus der unmodified List eine Seite gewählt.

2.9.4. Gemeinsamer Speicher (500)

- Es wird ein gemeinsamer Hauptspeicherbereich für mehrere Prozesse eingerichtet
- Ladeoptimierung: Mehrfach verwendeter Code wird nur einmal in den physischen Speicher geladen, dort von mehreren Prozessen gelesen. Bei Änderung der Daten wird eine private Kopie der Daten für den schreibenden Prozess erstellt.

3. Benutzerinteraktion aus Systemsicht (359)

3.1. Programmiermodell (367)

- Programmgesteuerter Ablauf: Das Programm bestimmt, wann Eingaben zu erfolgen sind. Es existiert ein fester Ein-/Ausgabeablauf (linear).
- Ereignisgesteuerter Ablauf: Die Reihenfolge der Ein-/Ausgabe wird vom Benutzer beeinflusst. Programmablauf wird durch Eingaben bestimmt, also nicht mehr linear.

3.1.1. Thread-Arten (384)

- GUI-Thread: Realisieren einen ereignisgesteuerten Programmablauf. Primary Thread ist winmain().
- Konsole-Thread: Realisieren einen programmgesteuerten Ablauf.
- Worker-Thread: Keine Interaktion mit Benutzer, führen für gewöhnlich umfangreichen Code aus.

3.2. Windows-GUI (383)

3.2.1. Verarbeitung von Benutzereingaben (384)

Eingaben von Benutzer über Maus / Tastatur werden vom Betriebssystem als „Ereignis“ erkannt und in Form von Ereignismeldungen an die entsprechende Applikation dirigiert.

3.2.2. Ereignisverarbeitung im Programm (385)

Die Grundstruktur eines Programms besteht aus zwei Teilen: GetMessage() entnimmt eine Meldung aus der Ereignis-Warteschlange und DispatchMessage() ruft die passende Fensterprozedur auf. Eine Fensterprozedur verarbeitet die Ereignisse für eine bestimmte Fensterklasse. Diese Fensterklasse beschreibt Grundeigenschaften eines bestimmten Fenstertyps und besitzt genau eine Fensterprozedur.

3.2.3. Ereignis-Meldungstypen (387)

Jede Ereignismeldung ist einem bestimmten Meldungstypen (WM_PAINT, WM_SIZE, ...) zugeordnet. Für jeden Meldungstyp ist die genaue Bedeutung der zwei Meldungsparameter gesondert festgelegt.

3.2.4. Meldungsübermittlung (391)

- Gepuffert: Meldung wird in Meldungs-Warteschlange geschrieben
  - Benutzer-Thread: Gepufferte Übertragung mittels PostMessage().
- Direkt: Direkter Aufruf der Fensterprozedur durch Betriebssystem. Warteschlange wird umgangen.
  - Benutzer-Thread: Direkte Übertragung mittels SendMessage(). Sender wird solange blockiert, bis der Empfänger-Thread bereit zum Meldungsempfang ist!

3.2.5. Tastaturmeldungen (393)

Tastaturmeldungen gehen immer an dasjenige Fenster, das den Fokus hat. Die WM_CHAR-Meldung muss in der Meldungsschleife aus der WM_KEYDOWN-Meldung erzeugt werden. Dies wird durch den Aufruf von TranslateMessage() erreicht. WM_CHAR returniert einen ASCII-Code für Tasten.

3.2.6. Fensterhierarchie (394)

- Kindfenster: Wird auf die Fläche des Elternfensters begrenzt. Verschwindet nie hinter Elternfenster.
- Besitzerfenster: Kann über Fläche des Besitzers hinausragen. Verschwindet nie hinter Besitzer.
- Z-Order: Imaginäre dritte Koordinatenachse. Entspricht der „Ebene“ des Fensters.

3.3. UNIX-GUI (363 / 370)

Der Unix-Kern selbst enthält keine Unterstützung grafischer Bedienoberflächen, sondern basiert auf rein textorientierter Ein-/Ausgabe. Die übliche Lösung besteht darin, entsprechende Software auf den Kern aufzusetzen.

3.3.1. X-Window-System

- Bildet die unterste Schicht. Es stellt Basisfunktionen zum Zeichnen von Punkten, Rechtecken usw. zur Verfügung.
- X selbst legt die Gestaltung der Bedienoberfläche nicht fest, sondern überlässt dies übergeordneten Softwareschichten.
- X teilt sich in einen X-Server und mehrere X-Clients auf. Der X-Server ist für die Visualisierung und die Benutzereingaben verantwortlich, die X-Clients stellen die Applikationen dar.
- X-Clients können auch auf Computer laufen, die über das Netzwerk mit einem X-Server kommunizieren.

3.3.2. X-Grundlagen (373)

X-Server und X-Client kommunizieren mittels des X-Protokolls. Man unterscheidet zwischen vier Typen von Paketen:

- Requests: Dienstanforderung (Client -> Server). Diese werden lokal gepuffert und nur dann gesendet, wenn er Client auf ein Ereignis zu warten beginnt, welches der X-Server auslösen muss.
- Replies: Antworten auf Anforderungen (Server -> Client)
- Events: Spontante Ereignismeldungen (Server -> Client). Ereigniss werden an beiden Enden gepuffert. Jeder Client spezifiziert, welche Ereignistypen er vom Server empfangen will. Nicht empfangene Ereignisse werden an das übergeordnete Fenster weitergeleitet. Die Events werden nach dem FIFO-Prinzip vom Client verarbeitet. Mehr zur Ereignisbehandlung auf Seite 378-383.
- Errors: Fehlermeldungen (Server -> Client)

3.3.3. Fenster-Grundlagen (375)

- Das Fensterkonzept ist einfach gehalten. Innerhalb eines Grundfensters (root window), das in der Regeln den gesamten Bildschirm einnimmt, können weitere Fenster (children) definiert werden. Siblings sind Kindfesnter des gleichen Elternfensters.

3.3.4. Desktop-Manager

- Da X nur eine sehr einfache Ereignisbehandlung und Grafikroutinen zur Verfügung stellt, wird es durch eine darüber liegende Softwareschicht ergänzt, auch Desktop-Manager genannt.
- Liefert eine elementare Benutzeroberfläche für Dateimanipulationen, Drag-and-Drop, Taskleiste und ähnlichem.
- In der Regel gehört auch ein ganzes Bündel von kleineren Anwendungsprogrammen (Calc, Textedit, ...) dazu.

3.3.5. Window-Manager (371)

- Um ein vollständiges GUI mit einer üblichen Funktionalität zu erhalten, wird noch eine dritte Komponente benötigt: der Window-Manager.
- Er ist für die Koordination (Verschiebung, Minimierung, ...) der Applikationsfenster, aber auch für eine bestimmte Art der Darstellung umrandeter Fenster mit Knöpfen verantwortlich.
- Der WM beinflusst das Aussehen der Fenster, indem er in jedem Top-Level-Fenster ein Extrafenster plaziert.
- Der WM ist einfach ein weiterer X-Client mit Sonderrachten für Fensterverwaltung.
- Unter KDE wird kwin, unter Gnome metacity als WM verwendet.

4. Ein- und Ausgabe (311)

Im Von-Neumann-Modell werden unter Peripherie alle Geräte verstanden, die über den Funktionsblock Input/Output angesprochen werden.

4.1. Programmgesteuerte E/A -> Polling (313)

Merkmal dieses Ansatzes ist die dauerende Programmkontrolle über den E/A-Vorgang. Dazu gehören auch etwaige Wartezeiten, wenn eine Operation des Peripheriegeräts einige Zeit dauert. Zu diesem Zweck werden Warteschleifen benutzt.

- Vorteil: Einfachheit des Ansatzes
- Nachteil: Es wird sehr viel Rechenzeit verschliessen durch „Busy-Waits“. Im Worst-Case kann dies auch zur völligen Blockierung des OS infolge exzessiver Wartezeiten führen.

4.2. Interruptgesteuerte E/A (314)

Die wesentliche Verbesserung gegenüber der programmgesteuerten Ein-/Ausgabe ist die Nutzung der Wartezeiten für andere Aufgaben. Dies wird durch das Unterbrechungssystem des Prozessors ermöglicht. Für eine E/A-Aktion wird das Peripheriegerät, bzw. sein Controller beauftragt. Die Vollendung der E/A meldet das Peripheriegerät über ein HW-Interrupt-Signal an die CPU. Die CPU aktiviert darauf hin eine Behandlungsroutine in der Software.

4.2.1. Unterbrechungssursachen (315)

- Fehlsituation: Fehler bei Rechenoperationen (Div / 0), ungültiges Maschinenbefehlsformat, Adressfehler, Fehler im Bussystem
- SW-Interrupt: Ausgelöst durch ein Programm mittels eines speziellen Maschinenbefehls (trap instruction)
- HW-Interrupt: Eine Peripherieeinheit meldet über ein HW-Signal ein Ereignis an die Software.

4.2.2. Synchrone / Asynchrone Unterbrechungen (317)

- Synchrone Unterbrechnungen: Ihr Auftreten ist an die Ausführung einer Instruktion gebunden. Ihr Zeitpunkt ist also vorher bestimmbar. Beispiele sind: Trap-Befehle, Div / 0, Überlauf.
- Asynchrone Unterbrechungen: Ihr Auftreten ist nicht an den Programmablauf gebunden. Ihr Zeitpunkt ist also nicht voraussagbar. Beispiele sind: Busfehler, Interrupts von Peripheriegeräten.

4.2.3. Priorisierung von Unterbrechungen (318)

- Einebenen-Unterbrechungssystem: Eine akzeptierte Unterbrechung wird vollständig abgehandelt, eine weitere Interrupt-Anforderung kann während dieser Zeit nicht behandelt, jedoch aber gespeichert werden (FIFO).
- Mehrebenen-Unterbrechungssystem: Später auftretende Unterbrechungsanfoderungen werden beantwortet, wenn sie eine höhere Priorität haben als die gerade laufende Interrupt-Serviceroutine.

4.3. E/A mittels DMA (321)

Beim DMA-Betrieb (Direct Memory Access) geht es darum, dass Teile von E/A-Vorgängen anstatt durch die CPU durch einen Peripheriekontroller erledigt werden. Genau genommen handelt es sich hierbei stets um Datentransfers. Damit der DMA-Kontroller im DMA-Bertrieb die Daten selbständig aus dem Speicher holen kann, muss er den Prozessorbus übernehmen können.

4.3.1. DMA-Betriebsarten (322)

- Einzeltransfer: Auch Cycle-Steal-Mode genannt. Auf eine externe Anforderung hin wird ein einzelnes Byte / Wort / Langwort übertragen. Dazu wird der Bus für ein paar wenige Zyklen der CPU gestohlen. Bis zur nächsten externen Anforderung wird der Bus jedoch an die CPU zurückgegeben. Damit kann im Hintergrund ein Peripheriegerät mit kleiner Datenrate bedient werden.
- Blocktransfer: Auch Burst-Mode genannt. Alle Daten des Transfers werden unmittelbar hintereinander übertragen. Während dieser Zeit steuert der Peripheriekontroller den Bus dauernd, die CPU hat also keinen Buszugriff, bis alle Daten übertragen sind. Ein Anwendungsbeispiel könnte die Übertragung eines Datenblocks von der HD in den Memory sein.

4.3.2. Direkter/Indirekter Datentransfer (323)

- Direkter Datentransfer: Hierbei signalisiert das Peripheriegerät dem DMA-Kontroller die Bereitschaft für den Transfer. Die Daten werden anschliessend direkt vom Peripheriegerät in den Hauptspeicher geschrieben, ohne den Umweg über den DMA-Kontroller zu nehmen.

- Indirekter Datentransfer: Beim indirekten Datentransfer liest der DMA-Kontroller den Datenwert aus einem Datenregister des Peripheriegeräts aus und schreibt es anschliessend in den Hauptspeicher. Der DMA-Kontroller wird also als Brücke verwendet, wodurch immer zwei Buszyklen (Lesen, Schreiben) benötigt werden.

4.3.3. DMA-Betriebsphasen (324)

- Initialisierung: Der DMA-Kontroller wird programmiert (Quell- und Zieladresse, Anzahl Byte, ...)
- DMA-Betrieb: Der Kontroller übernimmt je nach Betriebsart (Einzel- oder Blocktransfer) teilweise oder dauerthaft den Bus. Der Prozessor ist in dieser Phase abgekoppelt, es findet der eigentliche Datentransfer statt.
- Abschluss: Die CPU wird mittels Interrupt darüber informiert, dass alle Daten transferiert wurden.

4.4.Eingabe/Ausgabesystem

Das Betriebssystem stellt logische Kanäle zwischen Anwendungsprogrammen und der Hardware dar.

4.4.1. Treiber (325-328)

- Als Treiber bezeichnet man diejenige Softwareteile, welche die eigentliche Verbindung zwischen den Anwenderprozessen und den Geräten, bzw. deren Peripheriekontrollern schaffen.
- Um unterschiedliche Geräte zu unterstützen, zu denen auch solche zählen, die bei der Programmierung des OS noch unbekannt waren, ist eine Treiberschnittstelle definiert.
- Moderne OS benutzen eine eigentliche Treiberhierarchie. Diese enthält einen logischen und physischen Teil. Der logische Teil unterstütz Funktionen, die für die gleiche Geräteart einheitlich gelöst werden können, der physische Teil enthält dann nur noch die gerätespezifischen Teile.

4.4.2. E/A-Schnittstelle (329)

Für die Entwicklung von Anwendenderprogrammen steht eine weitgehend geräteunabhängige Programmierschnittstelle zur Verfügung. Die Umsetzung der standardisierten E/A-Operationen in gerätespezifische Befehle wird dabei durch die Treibersoftware erbracht.

4.4.3. E/A-Pufferung

Grundsätzlich ist es möglich, EA-Daten zwischen einem Benutzerprozess und einem Peripheriegerät direkt durchzureichen. Meistens ist jedoch eine Zwischenpufferung innerhalb des OS von Vorteil. Dabei wird zwischen 4 Methoden unterschieden: Keine Pufferung, Einfache Pufferung, Doppelte Pufferung und Zirkuläre Pufferung.

5. Programmentwicklung (573)

5.1. Programmübersetzung (574)

Bei der Programmübersetzung wird zwischen Mehrschritt-Übersetzung und Einschritt-Übersetzung unterschieden. Bei letzterem führt ein Programm alle Schritte der Mehrschritt-Übersetzung aus. Dadurch vereinfacht sich die Programmübersetzung für den Endanwender.

5.1.1. Präprozessor (577)

Der Präprozessor führt eine Vorverarbeitung des C/C++ Quellcodes. Er ersetzt dabei #define | #include mit den zugehörigen Inhalten und verarbeitet Anweisungen der bedingten Kompilierung.

- gcc –E test.c -> File nur mit dem Präprozessor verarbeiten, Dateiendung .i

5.1.2. Compiler (575)

Der Compiler übersetzt ein Quellprogramm aus einer High level Language (HLL) in Assemblersprache.

- gcc –S test.c -> Erzeugt Assembler-Code-Datei (Dateiendung .s)

5.1.3. Assemblierung (575)

Der Assembler übersetzt ein Assembler-Quellprogramm in relozierbaren, nicht ausführbaren Objektcode (Maschinencode und Zusatzinformationen). Dateiendung .o.

- gcc –c test.c -> Erzeugt eine Objekt-Datei

5.1.4. Binder / Linker (576)

Der Binder fügt getrennte übersetzte Programmteile und bereits vorhandene Bibliotheksfunktionen zu einem Gesamtprogramm zusammen, wobei das Resultat eine reolzierbare Objektdatei oder ein absolut ladbarer, dh. Ausführbarer Code sein kann. Ausserdem werden Code- und Datenteile der Eingangsdateien so angeordnet, dass sie im Adressraum nicht überlappen.

5.2. T-Notation (578)

Für die Darstellung der Übersetzungsvorgänge und die Charakterisierung der Werkzeuge eignen sich die so genannten T-Diagramme besonders gut.

5.2.1. Residente Werkzeuge vs Cross Tools (579)

Residente Werkzeuge erzeugen Code für den Rechner, auf dem sie selbst ablaufen. Die Cross Tools hingegen erlauben die Codeerzeugung für andere Rechnerplattformen.

5.3. Automatisierte Übersetzung (581)

Um ganze Applikationen effizient zu übersetzen, die den Quellcode über eine Vielzahl von Dateien verteilt haben, ist das Dienstprogramm make ein beliebtes Automatisierungswerkzeug. Es berücksichtigt, dass oft nur Änderungen an einem kleinen Teil der Quellcodedateien durchgeführt werden und übersetzt nur diese Dateien neu.

5.4. Adressraumbelegung und Relokation (585)

5.4.1. Sektionen (586)

- Text Section (.text): Enthält den Programmcode und konstante Daten
- Data Section (.data): Enthält initialisierte globale Variablen und initialisierte statische (lokale) Variablen
- Base Section (.bss): Enthält uninitialisierte Daten (globale oder lokale statische Variablen, Stapel).

5.4.2. Relokation (587)

Under der Relokation von Programmen versteht man das Abändern von Zugriffsadressen im Code von Programmen.

- Relokationsbedarf bei der Programmübersetzung: Bei der Erzeugung der ausführbaren Datei werden alle Sektionen aller gebundenen Objektmodule nebeneinander, d.h. nicht überlappend im Adressraum platziert (=Bindevorgang)
- Relokationsbedarf beim Laden des Programms: Dies ist nötig, falls ein Programm nicht auf der vorgesehen Adresse gestartet werden kann, für die es übersetzt wurde.