

Betriebssysteme

1 Einführung.....	4
1.1 Was ist ein Betriebssystem.....	4
1.1.1 Einordnung im Rechnersystem	4
1.1.2 Aufgaben eines Betriebssystems	5
1.1.3 Klassifikation von Betriebssystemen.....	5
1.1.4 Komponenten und Grundkonzepte.....	6
1.1.5 Betriebsziele	7
1.2. Historischer Überblick	7
1.3 Hardwaregrundlagen	8
1.3.1 Prozessor	8
1.3.2 Speicher	8
1.3.3 Ein/ Ausgabe	8
1.3.4 Bussysteme.....	8
1.4 Strukturierung von Betriebssystemen	9
1.4.1 Monolithische Systeme	9
1.4.2 Geschichtete Systeme.....	9
1.4.3 virtuelle Maschinen	9
1.4.4 Client/ Server – Architekturen	9
1.4.5 Microkernel	9
2 Prozessverwaltung.....	10
2.1 Prozesse	10
2.1.1 Lebenszyklus eines Prozesses	10
2.1.2 Prozesshierarchie.....	11
2.1.3 Zustände	11
2.1.4 Prozesswechsel.....	11
2.1.5 Prozessverwaltung.....	12
2.2 Scheduling	12
2.2.1 Grundlagen	12
2.2.2 Scheduling im Stapelverarbeitungsbetrieb	13
2.2.3 Scheduling bei interaktiven Systemen	14
2.2.4 Echtzeit-Scheduling	15
3 Synchronisation.....	17
3.1 Race Conditions und kritische Abschnitte	17
3.2 Reine Softwarelösungen.....	17
3.2.1 Ein erster Fehlversuch	18
3.2.2 Strict Alternation	18
3.2.3 Peterson-Algorithmus.....	18
3.3 Verfahren mit Unterstützung der Hardware.....	19
3.3.1 Sperren der Interrupts.....	19
3.4 Verzicht auf aktives Warten	20
3.4.1 Semaphore	20
3.4.2 Mutexe.....	21
3.5 Synchronisationsprobleme	21
3.5.1 Erzeuger-Verbraucher-Problem	21
3.5.2 Leser-Schreiber-Problem	22

4 Verklemmung.....	24
4.1 Betriebsmittel	24
4.2 Überblick.....	24
4.2.1 Bedingungen.....	24
4.2.2 Belegungs-Anforderungs-Graph	25
4.2.3 Behandlung von Verklemmungen.....	25
4.3 Erkennen und Beheben von Verklemmungen.....	25
4.3.1 Erkennung bei einem BM pro Klasse	25
4.3.2 Erkennung bei mehreren BM pro Klasse	26
4.3.3 Wann wird Erkennung ausgeführt.....	26
4.3.4 Behebung von Verklemmungen.....	26
4.4 Verhindern von Verklemmung.....	27
4.4.1 Wechselseitiger Ausschluss	27
4.4.2 Behalten und Fordern	27
4.4.3 Nicht-Entziehbarkeit	27
4.4.4 Zirkuläres Warten.....	27
4.4.5 Bewertung	28
4.5 Vermeidung	28
4.5.1 Beschreibung des Systems	28
4.5.2 Sichere und unsichere Zustände	28
4.5.3 Bankier's Algorithmus	28
4.5.4 Bewertung	29
4.6 Ignorieren	29
5 Hauptspeicherverwaltung.....	30
5.1 Einfache Speicherverwaltung.....	30
5.1.1 Adressierung.....	30
5.1.1.1 Direkte Adressierung.....	30
5.1.1.2 Relative Adressierung	31
5.1.2 Schutz des Speichers	31
5.1.3 Aufteilung des Hauptspeichers.....	31
5.1.3.1 Aufteilung bei Einprozessor-Systemen	31
5.1.3.2 Aufteilung bei Mehrprozessor-Systemen.....	31
5.1.4 Zuteilung	32
5.1.4.1 Datenstrukturen	32
5.1.4.2 Strategien.....	33
5.1.4.3 Fragmentierung und Kompaktierung	33
5.2 Swapping.....	34
5.3 virtueller Speicher	34
5.3.1 Paging.....	34
5.3.1.1 virtuelle Adressen.....	35
5.3.1.2 Seitentabelle	35
5.3.2 Seitenersetzungsstrategie	36
5.3.2.1 Globale und lokale Seitenersetzung	36
5.3.2.2 Not Recently Used (NRU)-Algorithmus.....	36
5.3.2.3 First-In, First-Out (FIFO)-Algorithmus	37
5.3.2.4 Second-Chance-Algorithmus	37
5.3.2.5 Clock-Algorithmus.....	37
5.3.2.6 Clock-Algorithmus mit zwei Zeigern	38
5.3.2.7 Least Recently Used (LRU)- Verfahren.....	38

Erstellt von Oliver Fleck

Fach: Betriebssysteme

Dozent: Herr M. Friedmann

Zusammenfassung Semester III

5.3.2.8 Not Frequently Used (NFU)- Verfahren	38
5.3.2.9 NFU mit Aging.....	38
5.3.2.10 Zeitpunkt des Einlagerns	39
5.3.2.11 Thrashing.....	39
5.3.2.12 Working-Set-Verfahren.....	39
5.3.3 Weitere Aspekte	39
5.3.3.1 Gemeinsam genutzte Speicherseiten	39
5.3.3.2 Vorrat an freien Seitenrahmen	40
5.4 Segmentierung.....	40
5.4.1 Vorüberlegung.....	40
5.4.2 Adressierung und Segmente.....	40
5.4.3 Realisierung ohne virtuelle Adressräume	40
5.4.4 Realisierung mit virtuellen Adressräumen	41
6 Ein- /Ausgabe	42
6.1 E/A-Hardware	42
6.1.1 Geräteklassen	42
6.1.2 Einbindung	42
6.1.3 Kommunikation CPU ↔ Gerät	42
6.1.3.1 Ein-/Ausgabebefehle	43
6.1.3.2 Memory-Mapped-I/O	43
6.1.3.3 Interrupts	43
6.1.3.4 Direkt Memory Acces	43
6.2 E/A-Software.....	43
6.2.1 Zugriff auf Hardware	43
6.2.1.1 Programmed I/O.....	44
6.2.1.2 Interrupts	44
6.2.1.3 DMA.....	44
6.2.2 Schichtenmodell	44
6.2.2.1 Interrupt-Handler.....	44
6.2.2.2 Gerätetreiber.....	44
6.2.2.3 Geräteunabhängige Software	44
6.2.3 Programmierschnittstelle für Benutzerprogramme	45

1 Einführung

1.1 Was ist ein Betriebssystem

nach DIN 44300:

- gebildet durch Programme eines digitalen Rechensystems
- bilden Grundlage für mögliche Betriebsarten des digitalen Rechensystems
- steuert und überwacht Programme
- verwaltet Ressourcen
- stellt Umgebung zur Verfügung in der andere Programme ausgeführt werden können

1.1.1 Einordnung im Rechnersystem

Das Betriebssystem ist als Schicht zwischen Hardware und (Benutzer-) Programmen zu sehen:

Anwendungsprogramme Textverarbeitung Flugbuchung Spiele	User Mode
Systemprogramme Compiler Editor Werkzeuge Shell	System Call
Betriebssystem	Kernel Mode
Hardware	

- GUI und Shell **nicht** Teil des eigentlichen Betriebssystems.
- Trennung zwischen Betriebssystem und den weiteren Systemprogrammen. Diese werden zwar häufig mit dem BS ausgeliefert, sind aber nicht Teil des BS!
- BS schirmt die Hardware vollständig von den Benutzerprogrammen ab. Hierzu benötigt das BS Hilfe der Hardware und der CPU.
- Meist Unterteilung in verschiedene Modi:
 - o **privilegierter Modus** (kernel mode, supervisor mode)
 - o **eingeschränkter Modus** (user mode)

Der privilegierte Modus erlaubt **direkte Zugriffe** auf die Hardware, um diese einzurichten.
 Den Benutzer- und Systemprogrammen steht **nur** der **eingeschränkte Modus** zur Verfügung!
- Um auf die Hardware zuzugreifen muss ein Benutzerprogramm die Dienste des BS verwenden. Da diese Funktionen nur im priv. Modus ausgeführt werden können muss in diesen gewechselt werden. Dies ist Benutzerprog. aus Sicherheitsgründen nicht unkontrolliert gestattet. Hierzu steht Schnittstelle bereit:
 Diese Systemaufrufe werden meist über Software-Interrupts, die durch Maschinenbefehle (Taps) ausgelöst werden, durchgeführt.
 Die direkte Nutzung der „Taps“ ist meist unhandlich, da die Parameter mittels Maschinensprache gezielt platziert werden müssen!

1.1.2 Aufgaben eines Betriebssystems

Zentrale Aufgaben:

Abstraktion der Hardware:

- vereinfachter Zugriff
- verstecken der technischen Details
- vereinheitlichte Nutzung der Geräte

Verwaltung der Betriebsmittel:

- sowohl Hardware- als auch Software-Betriebsmittel
- zeitliche Aufteilung (z.B. Prozessor, Drucker, ...)
- räumliche Aufteilung (z.B. Hauptspeicher, Plattenplatz, ...)
- Schutzmassnahmen

Bereitstellung von Diensten, z.B.:

- Ausführen von Programmen
- Kommunikation
- Synchronisation
- Ein/ Ausgabe (direkter Zugriff auf Hardware ist dem Benutzer verwehrt!)

1.1.3 Klassifikation von Betriebssystemen

1) Einprozess-Systeme:

- keine Parallelverarbeitung
- einfache Strukturierung des BS

2) Mehrprozess-Systeme:

- scheinbar parallele Ausführung mehrerer Prozesse

3) Einbenutzersysteme:

- nur ein Benutzer gleichzeitig
- trivialer Fall: Einbenutzer/ Einprozess-Systeme (z.B. DOS, CPM)

4) Mehrbenutzersysteme:

- mehrere Benutzer **gleichzeitig**
- aufwändigere Verwaltung der Betriebsmittel
- Abgrenzung der Benutzer voneinander

Man kann diese Betriebsarten noch weiter Einteilen:

Einzelauftragsverarbeitung:

- Ein Benutzer hat Kontrolle über gesamtes System
- Manuelles Laden und Ausführen eines Programms
- Schlechte Ausnutzung der Ressourcen (CPU liegt während E/ A brach)

Stapelverarbeitung:

- Mehrere Jobs werden gesammelt und dann als Stapel verarbeitet (first in – first out)
- Keine Interaktion mit Benutzer
- Lange Wartezeiten für Benutzer; langer Prozess blockiert alle anderen hinter ihm
- Schlechte Ausnutzung der Ressourcen
- Verbesserung: Spooling (puffern der Ein- und Ausgabe, so dass keine Wartezeiten durch E/ A entstehen)

- z.T. Multiprogramming: Umschaltung zwischen mehreren Programmen zur Optimierung der Systemleistung und Systemauslastung

Realzeitverarbeitung:

- Multiprogramming mit erweiterter Anforderung
- Steuerung zeitkritischer Prozesse
- Einhalten der Zeitschranken für wiederkehrende Prozesse
- Einhaltung von Reaktionszeiten für externe Ereignisse

Dialogverarbeitung:

- interaktives Arbeiten mit dem System
- kurze Antwortzeiten
- zwei wesentliche Ausprägungen:
 - w **Timesharing**
 - interaktiver Betrieb mit Multiprogramming
 - jeder Benutzer hat scheinbar die Maschine für sich (jedoch mit langsamerer Ausführungszeit)
 - w **Transaktionssysteme**
 - gemeinsame Nutzung eines Datenbestandes
 - keine individuellen Programme pro Nutzer, stattdessen gemeinsames System

1.1.4 Komponenten und Grundkonzepte

Konzepte haben meist zwei Anteile:

- **Mechanismus:** Beschreibt, wie bestimmte Aufgaben erledigt werden sollen
- **Strategie:** Beschreibt wann und zu welchem Zweck Mechanismen zum Einsatz kommen

Der Mechanismus ist meist leicht zu beschreiben und bei vielen BS ähnlich. Jedoch gibt es eine breite Auswahl an Strategien mit jeweiligen Vor- und Nachteilen. Manche BS erlauben die Auswahl der Strategie durch Benutzer!

Komponenten sind z.B.:

- **Prozesse:**
 - § Programm in Ausführung
 - § Prozesswechsel
 - § Kommunikation
 - § Synchronisation
 - § Verklemmung
- **Dateisysteme:**
 - § dauerhafte Datenspeicherung
 - § Erzeugen, Löschen, Verwalten, Lesen, Schreiben, ... von Dateien
 - § Organisation: i.A. Verzeichnisse
 - § Sicherheit
 - § Konsistenz
- **Speicherverwaltung:**
 - § Bereitstellung von Speicher für Programme
 - § versch. Adressräume: virtuell (System-Auslagerungsdatei)/ physikalisch
 - § Speicherschutz

- **Ein/ Ausgabe:**
 - § Abstraktion von den Geräten
 - § Zuteilung an Prozesse
- **Benutzerverwaltung:**
 - § Grundlage für weitere Schutzmassnahmen
 - § Rechtevergabe
- **Benutzeroberfläche:**
 - § nicht Teil des BS, jedoch wichtiges Bindeglied zwischen Benutzer und System

1.1.5 Betriebsziele

Abhängig vom Anwendungszweck ergeben sich verschiedene Ziele:

- hohe Systemauslastung
- kurze Reaktionszeiten
- Einhalten der Zeitschranken

1.2. Historischer Überblick

Generation: 1945-55

- Relais, Röhren, Steckkontakte, später Lochkarten
- Einzelauftragsverarbeitung
- Kein BS
- Programme komplett Maschinensprache
- Keine Entwicklungstools
- Enge Bindung Erbauern, Programmierern und Anwendern der Anlage
- Hauptsächlich numerische Berechnungen

Generation: 1955 – 65

- Transistoren, Ringkernspeicher, Magnetbänder
- Stapelverarbeitung
- Kein Parallelbetrieb: schlecht CPU – Auslastung während E/A
- Einfache Steuersprachen (JCL – Job Control Language)
- Erste Programmiersprachen & Compiler
- Kein interaktives Arbeiten

Generation: 1965 – 80

- ICs, Trommelspeicher, später Magnetplatten
- Bessere CPU – Auslastung durch Multiprogramming und Spooling
- Später: Dateisysteme
- Später: Timesharingssysteme mit interaktivem Betrieb

Generation: 1980 – 95

- Mikroprozessoren, anfangende Vernetzung der Rechner
- Erster PC, kleine Systeme für einzelne Benutzer
- Graphische Benutzeroberfläche

Generation: 95 – heute

- Vernetzung, Multimedia, mobile Systeme, usw. (aber ich denke mal da wisst ihr so einigermaßen Bescheid ☺)

1.3 Hardwaregrundlagen

1.3.1 Prozessor

- sequentielles Abarbeiten von Befehlen: Laden » Ausführen » Laden » Ausführen » usw.
- allgemeine Register für Daten
- Spezialregister: z.B. Programmzähler, Stackpointer, Statuswort
- verschiedene Modi mit unterschiedlichen Rechten
- Unterbrechung der sequentiellen Verarbeitung durch Interrupts

1.3.2 Speicher

Speicherhierarchie:

CPU – Register	geringe Kapazität, kurze Zugriffszeit, hohe Kosten
Cache	
Hauptspeicher	↓
Plattenspeicher	
Bänder, CDs	hohe Kapazität, lange Zugriffszeit, geringe Kosten

Zwei Sichtweisen:

- höhere Schichten bilden **Cache** für tiefere
- tiefere Schichten bilden **Hintergrundspeicher** für höhere

1.3.3 Ein/ Ausgabe

- i.A. über spezielle Controller-Bausteine mit Eigenintelligenz
- Controller verbirgt Details der Steuerung
- Kommunikation CPU » Controller:
 - § Memory-Mapped
 - § spezielle I/O-Befehle
- Kommunikation Controller » CPU
 - § Statusregister
 - § Interrupts
 - § DMA/ Busmastering

1.3.4 Bussysteme

- Kommunikation zwischen CPU, Hauptspeicher und anderer Hardware („Datenautobahn“ des PCs)
- häufig Kombination verschiedener Busse in einem System
- Unterscheidungskriterien für Busse:
 - § Geschwindigkeit
 - § Verwendungszweck
 - § Anzahl der Geräte am Bus

1.4 Strukturierung von Betriebssystemen

1.4.1 Monolithische Systeme

- alle Funktionen in einem Kern, keine weitere Abgrenzung
- innerhalb des Kerns weitere Ordnung
- zum Teil Verbesserung durch Modularisierung

1.4.2 Geschichtete Systeme

- verschiedene Funktionen bilden Schichten
 - jede Schicht stellt darüberliegenden Schicht Dienste zur Verfügung und nutzt Dienste der unteren Schicht
 - aufwändiges Durchlaufen der Schichten
- tiefe Schichten können nur auf wenig Funktionalität zurückgreifen

1.4.3 virtuelle Maschinen

- Bereitstellung virtueller Rechensysteme
- auf dem virtuellen System Ausführung eines BS
- mehrere BS auf einem Rechner zur gleichen Zeit

1.4.4 Client/ Server – Architekturen

- Systemdienste werden zu Klassen zusammengefasst
- Dienstklassen werden von Servern realisiert
- Server setzen auf Systemkern auf
- Flexibilität durch Austauschen der Server

1.4.5 Microkernel

- kleiner Kern mit reduzierter Flexibilität
- starke Trennung zwischen Mechanismus und Strategie: Strategie wenn möglich als Prozess im Benutzermodus
- i.d.R. nachrichtenorientierte Kommunikation

2 Prozessverwaltung

Der Prozess ist eines der wichtigsten Konzepte. Hiermit werden **Programme in Ausführung** beschrieben.

Hierbei wird zu jedem Programm ein eigener Prozess erzeugt, auch wenn dasselbe Programm mehrfach ausgeführt wird.

In der Regel sind die Ressourcen/ Betriebsmittel die für den Programmablauf benötigt werden mit dem jeweiligen Prozess verknüpft.

Gängige BS erlauben es, mehrere Prozesse scheinbar (bei Mehrprozessorsystem auch tatsächlich) gleichzeitig auszuführen. Dies wird als **Multitasking** bezeichnet.

Man unterscheidet zwischen **kooperativem** Multitasking (Prozess gibt freiwillig an anderen Prozess ab) und **verdrängtem** Multitasking (System entzieht laufendem Prozess den Prozessor und übergibt ihn einem anderen).

Der Teil des BS, der die Umschaltung vornimmt nennt sich **Dispatcher**.

Der **Scheduler** wiederum entscheidet welcher Prozess wann zur Anwendung kommt.

2.1 Prozesse

Durch Prozesse werden Programme sequentiell ausgeführt. Das bedeutet, dass das Programm als Folge von einzelnen Befehlen verstanden wird (auf Ebene des Prozesses gibt es zunächst keine Parallelität).

2.1.1 Lebenszyklus eines Prozesses

- wird erzeugt
- rechnet (meist mit Unterbrechungen)
- wird beendet

Erzeugen eines Prozesses durch z.B.:

- Systemstart
- Prozess startet Kinderprozess
- Benutzer startet Programm
- Batchjob wird im Hintergrund gestartet

In all diesen Fällen muss das System dazu einen neuen Prozess anlegen. Normalerweise stehen dazu spezielle Systemaufrufe bereit:

- Systemaufruf „fork“ bei UNIX
- Systemaufruf „CreateProzess“ bei Windows

Was erzeugt wird, sollte irgendwann auch beendet werden. Ursachen hierfür sind z.B.:

- Prozess hat seine Aufgabe zu Ende gebracht
- Prozess kann wegen Fehler nicht weiterrechnen
- Prozess verhält sich nicht systemkonform und wird vom System beendet
- Prozess wird von einem anderem terminiert

2.1.2 Prozesshierarchie

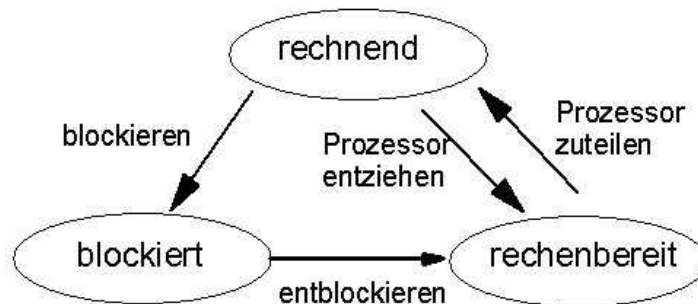
Da Prozesse durch Systemaufruf aus einem anderen Prozess heraus erzeugt werden entsteht eine Eltern – Kind – Beziehung.

In UNIX erfolgt diese Verwaltung sehr konsequent. Kindprozesse können ihre Herkunft erfragen.

Unter Windows hingegen verwaltet das System solche Beziehungen nicht, allerdings erhalten Elternprozesse vom System ein „Handle“, über das sie auf den Kindprozess zugreifen können.

2.1.3 Zustände

Während ihrer Ausführung wechseln Prozesse in der Regel ihre Zustände. Die Art und Anzahl der Zustände hängt stark vom jeweiligen System ab. Jedoch lassen sie sich auf 3 wesentliche Zustände reduzieren:



2.1.4 Prozesswechsel

Um zwischen mehreren Prozessen zu wechseln, werden vom BS folgende Schritte ausgeführt:

1. Sichern aller Informationen über den Stand des aktuellen Prozesses
2. Auswahl eines neuen Prozesses
3. Stand des neuen Prozesses wiederherstellen und diesen weiter ausführen

Diese Schritte sind stark von der Hardware abhängig. Teile des Dispatchers sind so hardwarenah, dass sie nur in Maschinensprache implementiert werden können.

Für Schritt 2 ist der Scheduler zuständig.

Prozesswechsel können bei verschiedenen Ereignissen erfolgen:

- Aufruf eines blockierenden Systemaufrufs:
Der aktuelle Prozess wird blockiert, ein anderer muss ausgewählt werden.
- Aufruf eines beliebigen Systemaufrufs:
System kann aktuellen Prozess rechenbereit setzen und anderen auswählen.
- Verarbeitung eines Interrupts:
Nach dem Interrupt kann System anderen Prozess ausführen.

Die ersten beiden Ereignisse sind synchron zur Ausführung des Prozesses, sie können sowohl bei verdrängten als auch bei nichtverdrängten Multiprogramming zum Einsatz kommen.

Der Prozesswechsel während asynchroner Interrupts hingegen sind Voraussetzung für verdrängendes Multitasking.

Prozesswechsel kosten i.A. viel Rechenzeit. Je nach System müssen zum Teil große Mengen an Verwaltungsinformationen (z.B. Speicherverwaltung) verarbeitet und geändert werden.

Zudem werden Inhalte des Prozessorcacher (wenn vorhanden) mit hoher Wahrscheinlichkeit ungültig. Auch hier kommt es zu erheblichen Geschwindigkeitseinbußen. Daher ist man bestrebt Prozesswechsel gering zu halten. Ist kein Prozess rechenbereit wird ein **Leerlaufprozess** ausgeführt.

2.1.5 Prozessverwaltung

Für die Prozesse zu verwalten verwendet BS eine Prozesstabelle die eine Datenstruktur für jeden Prozess anlegt (**PCB**, Process Control Block).

Diese werden in Warteschlangen verwaltet. Diese gibt es für:

- rechenbereite Prozesse; zum Teil mehrere Warteschlangen, je nach Schedulingstrategie
- jede Ressource, auf die Prozesse warten, um leicht einen Prozess auswählen zu können, sobald diese verfügbar ist.

Typische Infos, die im PCB gespeichert werden sind:

- Verwaltungsinfos: Besitzer (uid, gid), Prozesszeit, ...
- Prozesszustand
- Infos bei Prozesswechsel
- Infos über Prozesshierarchie (pid, ppid)
- Belegte Ressourcen
- Schedulinginfos

(diese Infos können je nach System stark variieren!)

2.2 Scheduling

Das Schedulingverfahren hat starken Einfluss auf Systemeigenschaften wie:

- Antwortzeit
- Auslastung der CPU
- Auslastung der I/O – Ressourcen

2.2.1 Grundlagen

Prozessverhalten:

- Prozesse rechnen eine bestimmte Zeit, bevor sie I/O – Operationen durchführen, während sie blockiert sind
- 2 Klassen von Prozessen:
 - i) CPU-lastig (computing-bound)
 - ii) I/O-lastig (I/O-bound)
 - iii) Unterscheidungskriterien: Länge der CPU-Belegung (CPU-burst) und Häufigkeit der I/O-Operationen

Zeitpunkt der Schedulingaktivierung:

- Prozesszeugung
- Prozessende
- Prozess wird blockiert
- Während Interruptbehandlung (Voraussetzung für verdrängendes Scheduling)

Mechanismus:

- nichtverdrängend: Scheduling nur, wenn Prozess blockiert wird (oder CPU freiwillig abgibt)
- verdrängend: Scheduling, wenn Prozess blockiert wird, oder bei Interrupt (z.B. wenn Prozess lange genug gelaufen ist)

Ziele beim Scheduling:

- Abhängig von Betriebsart!
- Generell:
 - i) Fairness
 - ii) Durchsetzung einer Systempolitik
 - iii) Gleichmäßige Auslastung des Systems

Ziele bei Stapelverarbeitung:

- Durchsatz (throughput) maximieren
- Verweilzeit (turnaround time) minimieren
- CPU-Auslastung maximieren

Ziele bei interaktiven Systemen:

- Antwortzeiten minimieren
- Verhältnismäßigkeit

Ziele bei Echtzeitsystemen:

- Einhalten von Zeitschranken
- Vorhersagbarkeit

2.2.2 Scheduling im Stapelverarbeitungsbetrieb

First-Come First-Served (FCFS):

- nichtverdrängendes Verfahren
- alle rechenbereiten Prozesse in Warteschleife
- vorderster Prozess ausgeführt, bis er blockiert
- sobald Prozess rechenbereit kommt er in die Schlange
- Vorteile:
 - i) Leicht zu verstehen
 - ii) Leicht zu implementieren
 - iii) Jeder Prozess irgendwann fertig
- Nachteile:
 - i) Bevorzugung CPU-lastiger Prozesse
 - ii) Oft lange Verweilzeit

Shortest Job First (SJF):

- nichtverdrängendes Verfahren
- rechenbereiter Prozess mit kürzester CPU-Belegung zuerst, bis er blockiert
- Problem: wie findet man diesen heraus?!?
- Vorteile:
 - i) Für feste Prozessmenge beweisbar optimale durchschnittliche Verweilzeit
- Nachteile:
 - i) „Verhungern“ (starvation) wenn kurze Prozesse immer wieder nachkommen

Shortest Remain Time Next:

- verdrängende Version von SJF
- wenn neuer Prozess aktiviert wird dessen CPU-Belegung kürzer ist verdrängt laufenden Prozess

- sonst wie SJF
- Vorteil:
 - i) Kurze Prozesse werden rasch ausgeführt
- Nachteil:
 - i) Problem des Verhungerns

Three-Level Scheduling

Bislang nur Schedulingstrategien von Prozessen, die auf CPU ausgeführt werden. Es gibt jedoch 2 weitere in Stapelverarbeitungssystemen:

Admission-Scheduling (long-term-Scheduling) zur Auswahl von neuen Aufträgen, die gestartet werden können. Hierbei wird die Auftragsreihenfolge nicht zwingend eingehalten.

Memory Scheduling (medium-term-Scheduling) verwendet um zu entscheiden, welche aktiven Prozesse zur Ausführung im Speicher gehalten werden und welche zeitweise auf Platte ausgelagert werden.

Ziel von Admission- und Memory-Scheduling ist es, immer eine Mischung von Prozessen am Laufenden zu halten, die das System optimal auslastet.

Speziell die Mischung aus CPU- und I/O-lastigen Prozessen spielt dabei eine wichtige Rolle. Memory- und CPU-Scheduling können in allen Arten von Systemen vorkommen, Admission-Scheduling kommt typischerweise nur bei Stapelverarbeitungssystemen vor.

2.2.3 Scheduling bei interaktiven Systemen

Ziel ist die Wartezeit gering zu halten. Dies hat erheblichen Einfluss auf die verwendeten Schedulingstrategien.

Die möglichen Strategien lassen sich zum Teil im Batchbetrieb oder bei Kombination der Betriebsarten einsetzen.

Round Robin (RR):

- verdrängend
- vorderster Prozess darf Rechnen
- Prozess nach Ablauf eines festen Zeitquantum verdrängt und ans Ende der Schlange
- Falls blockiert kommt der nächste dran
- Entblockierte Prozesse ans Ende der Warteschlange
- Prozesse die freiwillig abgeben kommen ebenfalls ans Ende der Schlange
- Problem: Wahl der Länge des Zeitquantums:
 - i) Zu kurz: viel Rechenzeit verloren
 - ii) Zu lang: schlechte Antwortzeiten
- Bewertung:
 - i) Alle CPU-lastigen Prozesse gleich viel Rechenzeit
 - ii) Alle Prozesse haben gleiche Priorität
 - iii) I/O-lastige Prozesse bekommen weniger Rechenzeit

Prioritätsgesteuerte Strategien:

- Verdrängend oder nichtverdrängend
- Jeder Prozess hat Priorität
- Prozess mit höchsten Priorität wird zuerst ausgeführt

- Prioritätsvergabe:
 - i) Statisch: Prozess erhält fester Priorität
 - ii) Dynamisch: Priorität vom Scheduler angepasst
- Problem: Prozesse mit hoher Priorität können theoretisch ewig laufen
Lösungsmöglichkeiten:
 - § langrechnende Prozesse herabstufen
 - § lange wartende Prozesse hochstufen (-> „Aging“)

Mehrere Warteschlangen:

- für rechenbereite Prozesse mehrere Warteschlangen, Zuordnung:
 - i) Priorität
 - ii) Betriebsart
- pro Warteschlange getrenntes Scheduling (nicht zwingend für jede Warteschlange dasselbe Verfahren!)
- Scheduling zwischen Warteschlangen:
 - § Prioritäten: Schlangen mit niedriger Priorität kommen nur dran wenn Schlangen mit hoher Priorität leer
 - § Zeitschreiben; pro Schlange verschiedener Anteil der CPU-Zeit
- Zuordnung von Prozessen zu Schlange:
 - i) Fest: Multi-Level-Queue-Scheduling (MLQ)
 - ii) Dynamisch: Multi-Level-Feedback-Queue-Scheduling (MLFQ)
- bei MLFQ: Kriterien für Wechsel der Schlange, z.B.:
 - § Herabstufen von Prozessen die Zeitscheibe ganz aufbrauchen
 - § Hochstufen von Prozessen die Zeitscheibe nicht aufbrauchen
 - § Hochstufen zeitkritischer Prozesse, etc.
- MLQF extrem flexibel parametrisierbar

Sonstige Verfahren:

1. Guaranteed Scheduling:

- Prozesse erhalten garantierten Anteil der Rechenzeit
- Problem der Verwaltung und falls Prozess lange schläft und plötzlich ganze Zeit anfordert

2. Fair Share Scheduling:

- pro Benutzer festes Rechenzeitkontingent, das auf Prozesse verteilt wird
- auch hier Verwaltungsprobleme

3. Lottery Scheduling:

- Zeitscheiben werden „verlost“
- Anzahl der Lose approximiert Anteil an Rechenzeit
- gezielte Weitergabe von Rechenzeit an andere Prozesse möglich
- Verhalten viel klarer vorhersagbar, als bei Prioritäten
- keine Garantien! Nur Wahrscheinlichkeiten

2.2.4 Echtzeit-Scheduling

Ein Echtzeitsystem ist System welches bestimmte Zeitverhaltengarantiert. Ziele sind:

- Einhalten der Zeiten für periodische Prozesse
- Einhalten der Reaktionszeiten für nichtperiodische Prozesse

Es werden zwei Typen von Echtzeitsystemen unterscheiden:

- **harte Echtzeit:** alle Zeitschranken müssen eingehalten werden
 - i) i.A. spezielle Hardware: z.B. keine HD, oft kein virt. Speicher
 - ii) spezielle BS
- **Weiche Echtzeit:** gelegentliches Nichteinhalten ist akzeptabel
 - i) z.T. in normalen NS integriert, z.B. in höchster Schlange beim MLFQ
- Im Echtzeitbetrieb zwei Arten von Prozessen: periodische & nichtperiodisch

Rate-Monotonic-Scheduling (RMS):

- verdrängend
- Anforderungen an kritische Prozesse:
 - i) Jeder Proz. Ist zyklisch mit fester Wiederholungsrate, d.h. Prozess muss in bestimmter Zeitschranke fertig sein
 - ii) Jeder Proz. Hat feste Verarbeitungszeit pro Zyklus
 - iii) Prozesse voneinander unabhängig
- Andere Prozesse dürfen nicht zeitkritisch sein und sind untergeordnet
- Prioritätsgesteuertes Scheduling mit statischen Prioritäten
- Wiederholungsrate ist Priorität

RMS **garantiert** Einhaltung der Zeitschranken, wenn die gemeinsame Prozessorauslastung aller zeitkritischen Prozesse kleiner als 69% ist.

Earliest Deadline First (EDF):

- verdrängendes Verfahren
- Scheduling nach Priorität: abhängig von Zeitschranke: wer zuerst fertig sein muss wird zuerst ausgeführt (macht Sinn!)
- Periodische Proz. Müssen pro Periode neu geplant werden
- Zeitschranken müssen bekannt sein
- Dynamische Prioritäten!
- Wird dringenderer Prozess rechenbereit verdrängt er den anderen
- Optimales Schedule: Voraussetzung CPU-Auslastung <100%

3 Synchronisation

Werden mehrere Prozesse oder Threads parallel ausgeführt, gibt es mehrere Gründe deren Ausführung zu koordinieren:

- kooperativer Zugriff auf gemeinsam genutzte Daten
- konkurrierender Zugriff auf Daten und Betriebsmittel
- Kommunikation zwischen den Prozessen
- Einhaltung einer vorgegebenen Reihenfolge

3.1 Race Conditions und kritische Abschnitte

Wird Ausführung der Prozesse nicht koordiniert kann es zu schwerwiegenden Programmfehlern kommen. Das Gesamtergebnis der Operationen ist dann von der Ausführungsreihenfolge der Prozesse abhängig. Dies bezeichnet man als „**race conditions**“.

Beispiel:

2 Prozesse sollen gemeinsam genutzte Variable ändern:

```
Prozess 1{                               Prozess 2{
Variable_einlesen();                     Variable_einlesen();
Variable_ändern1();                       Variable_ändern2();
Variable_speicher();                       Variable_speicher();}
```

Es gibt hierbei 4 mögliche Ergebnisse:

1. nur die Änderung aus Prozess 1 wird wirksam
2. nur die Änderung aus Prozess 2 wird wirksam
3. Änderungen werden in Reihenfolge Prozess 1 – Prozess 2 ausgeführt
4. Änderungen werden in Reihenfolge Prozess 2 – Prozess 1 ausgeführt

Die Abschnitte der beiden Prozesse sind **kritische Abschnitte** (critical section) in Bezug auf den Zugriff auf die gemeinsam genutzte Variable.

Kritische Abschnitte sollen in Bezug auf andere kritischen Abschnitte atomar (unteilbar) sein. Sie sollen entweder ganz oder gar nicht abgearbeitet werden. Das heißt jedoch nicht, dass das BS diesen Prozess nicht unterbrechen kann!

Eine Vermeidung von race conditions setzt 3 Forderungen voraus:

1. **Wechselseitiger Ausschluss** (Mutual exclusion): nur ein Prozess darf sich zu jedem Zeitpunkt innerhalb eines kritischen Abschnitts befinden.
2. **Fortschritt** (Progress): Kein Prozess der sich nicht im krit. Abschnitt befindet darf einen anderen blockieren.
3. **Eingeschränktes Warten** (Bounded Waiting): kein Prozess sollte unendlich lange warten müssen, bis er einen krit. Abschnitt betreten darf.

3.2 Reine Softwarelösungen

Zunächst werden Lösungen vorgestellt, die ohne Hilfe des BS auskommen und keine Unterstützung durch Hardware benötigen. Es werden ebenfalls keine privilegierten Befehle verwendet, so dass sie in beliebigen Anwendungsprogrammen einsetzbar sind. Es werden Sperrvariablen (lock variables) verwendet. Das Schreiben in den Hauptspeicher wird als atomar angenommen, d.h. es erfolgt ohne Unterbrechung.

3.2.1 Ein erster Fehlversuch

Erste Idee um gegenseitigen Ausschluss durchzusetzen ist eine gemeinsame Variable zu verwenden, die anzeigen soll ob der krit. Abschnitt frei oder belegt ist. Ein Prozess muss nun in einer Schleife die Variable so lange überprüfen, bis der Abschnitt frei ist. Bevor er nun den krit. Abschnitt betritt setzt er die Variable auf gesperrt, danach wieder auf frei:

```
while(lock!=0) { }  
lock=1;  
critical_region();  
lock=0;
```

Lösung **funzt nicht!** Wenn direkt nach while-Schleife zu einem anderen Prozess gewechselt wird, hat der erste den krit. Abschnitt schon betreten, aber die Sperre noch nicht gesetzt.

Somit kann der zweite Prozess ebenfalls den krit. Abschnitt betreten!

Der Zugriff auf Sperrvariable erfolgt nicht atomar. Vielmehr besteht er aus den beiden Teilen Auslesen und Setzen. Diese bilden zusammen selbst einen krit. Abschnitt. Auf diese Art ist das Problem nicht zu lösen!

3.2.2 Strict Alternation

Auch der nächste Ansatz benutzt nur normale Variable, um den gegenseitigen Ausschluss sicherzustellen:

Process1{ While(turn!=0) {} Critical_region0(); Turn=1; Noncritical_region{}; }	Process1{ While(turn!=1) {} Critical_region1(); Turn=0; Noncritical_region{}; }
--	--

Diese Lösung setzt den gegenseitigen Ausschluss dadurch, dass sie sicherstellt, dass sich die Prozesse immer abwechselnd im krit. Abschnitt befinden.

Bewertung:

- aktives Warten (busy waiting): die Prozesse verbrauchen während sie auf das freierwerden warten CPU-Zeit
 - Kriterium 2 verletzt: verlässt Prozess0 den k.A., kann er ihn erst dann wieder betreten, wenn Prozess1 ihn einmal betreten und wieder verlassen hat
 - Kriterium 3 verletzt: Prozess0 muss nach einmaligem Aufenthalt im k.A. ewig warten, falls Prozess1 den k.A. nie betritt
- ö **keine sinnvolle Lösung** zur Synchronisation!

Aktives Warten sollte generell vermieden werden, wenn nicht sichergestellt ist, dass nur kurz gewartet werden muss. Eine Sperre, die aktives Warten einsetzt wird als **spin lock** bezeichnet.

3.2.3 Peterson-Algorithmus

1981 wurde von G.L. Peterson ein leicht verständliches Verfahren für den gegenseitigen Ausschluss zweier Prozesse unter Einhaltung der oben angegebenen Bedingungen vorgestellt. Das Verfahren nutzt 2 Funktionen zum Betreten und Verlassen des k.A. und greifen auf Sperrvariablen zu. Vor Ausführung ist Variable turn unbekannt, die Werte der beiden Variablen interested ist 0.

Peterson-Algorithmus:

```
int turn;                // initialer Wert undefiniert
int interested[2];      // initial beide = 0
enter_region(int process) {
    int other=1-process;
    interested[process]=1;
    turn=process;
    while (turn==process && interested[other]==1) {}
}
```

```
leave_region(int process) {
    interested[process]=0;
}
```

Zum Absichern eines k.A. ruft der jeweilige Prozess vor Eintreten `enter_region` mit seiner Nummer (0 oder 1) auf, nach verlassen gibt er den k.A. mittels `leave_region` wieder frei.

Bewertung:

- Lösung erfüllt Bedingungen
- aktives Warten
- Lösung funktioniert so nur für 2 Prozesse

3.3 Verfahren mit Unterstützung der Hardware

Die folgenden Varianten nutzen spezielle Maschinenbefehle und Eigenschaften der Hardware, um den gegenseitigen Ausschluss sicherzustellen.

3.3.1 Sperren der Interrupts

Einfache Lösung zum Durchsetzen des gegenseitigen Ausschlusses ist es, beim Eintreten in einen k.A. die Interrupts zu sperren. Dadurch wird verhindert, dass der Prozessor entzogen werden kann.

Diese Lösung ist aus mehreren Gründen nicht generell einsetzbar:

- Benutzerprozesse dürfen keine Interrupts sperren, da sie sonst das System blockieren und ewig laufen können. Normalerweise können Interrupts nur im privilegierten Modus gesperrt werden.
- Bei Mehrprozessorsystemen können i.A. nur die Interrupts des Prozessors gesperrt werden, der den Prozess ausführt.

Dennoch ist das Sperren von Interrupts dann eine sinnvolle Methode, wenn innerhalb des BS-Kerns für einige Befehle der wechselseitige Ausschluss sichergestellt werden muss.

3.3.2 TSL-Anweisung

Viele Plattformen stellen spezielle Funktionen zum atomaren Auslesen und Setzen einer Sperrvariablen bereit. Dies soll an der TSL -(Test and Set Lock) Anweisung erläutert werden.

TSL-Anweisung hat folgendes Befehlsformat:

TSL Register, Speicheradresse

Bei Ausführung wird Inhalt der Speicherstelle in Register übertragen. Danach wird Wert auf 1 gesetzt. Diese Schritte erfolgen atomar. Bei Mehrprozessorsystemen werden zusätzlich

spezielle Steuerleitungen sichergestellt, dass auch die anderen Prozessoren im System die **Atomarität** nicht unterbrechen.

enter_region:

```
TSL Register, LOCK
CMP Register,0
JNE enter-Region
RET
```

leave_region:

```
MOVE LOCK,0
RET
```

Diese beiden Assembler-Unterprogramme werden vor Betreten bzw. nach Verlassen des k.A. aufgerufen.

Das Verfahren entspricht der fehlerhaften Implementierung aus Abschnitt 3.2.1 mit dem Unterschied, das Auslesen und Ändern der Sperrvariablen eine **atomare Einheit** bilden.

Bewertung:

- Bedingung 1-3 erfüllt
- Aktives Warten eingesetzt:
 - i) Sollte nicht innerhalb Benutzerprogramme verwendet werden, da CPU-Zeit sinnlos verbraucht
 - ii) Kann innerhalb BS-Kerns für kurze k.A. sinnvoll verwendet werden
- Standardwerkzeug zur Synchronisation von BS-Kern interner Funktionen auf Mehrprozessorsystemen

3.4 Verzicht auf aktives Warten

Die bisher vorgestellten Methoden alle Nachteil des aktiven Wartens => Rechenzeit vergeudet. Zudem bei prioritätsgesteuertem Scheduling kann Problem der Prioritätsinvertierung auftreten:

Prozess N mit niedriger Priorität betritt k.A. danach wird Prozess H mit hoher Priorität rechenbereit und bekommt Prozessor. Versucht H nun k.A. zu betreten, beginnt er aktiv zu warten. Da N niedriger wird dieser, solange H rechnet, nicht aktiviert und kann k.A. nicht verlassen. Dadurch verbleibt H endlos im aktiven Warten, ohne dass einer der Prozesse vorankommt.

Ansatz aktives Warten zu verhindern ist es Prozesse die auf Freiwerden eines k.A. warten solange zu blockieren, bis k.A. frei ist.

Einfache Lösungen, die Systemaufrufe zum programmgesteuerten Blockieren und Entblockieren verwenden, enthalten häufig race-conditions, die weiteres Synchronisieren erfordern.

Ein genereller Ansatz zur Lösung sind Semaphore, die von vielen BS bereitgestellt werden.

3.4.1 Semaphore

Ein Semaphor ist Zählvariable, die über 2 Funktionen angesprochen werden kann:

- down: Wert eines Semaphors dekrementiert. Hierbei 2 Fälle:
 - i) Wert >0: Wert um 1 verringert und Prozess läuft weiter
 - ii) Wert =0: Wert unverändert, Prozess blockiert, bis anderer Prozess up ausführt
- up: Semaphor inkrementiert. Auch hier 2 Fälle:

- i) am Semaphor kein Prozess blockiert: Wert um 1 erhöht
- ii) am Semaphor Prozess blockiert: anderer blockierter Prozess entblockiert, Wert bleibt gleich

Semaphore 1965 von E.W. Dijkstra vorgestellt. Wesentlicher Eigenschaft: up und down **atomar**.

Die Strategie mit der up-Funktion einen Prozess zum Entblockieren auswählt nicht festgelegt. Denkbar hier z.B. FiFo-Warteschlangen oder Prioritäten basierende Strukturen (z.B. Priority-Queues). Wesentlich ist, dass wartender Prozess beim Entblockieren immer übergangen werden darf, da sonst Bedingung 3 verletzt!

Bewertung:

- kein aktives Warten
- alle 3 Bedingungen erfüllt
- klare Semantik
- vielfältige Einsetzbarkeit, auch über wechselseitigen Ausschluss hinaus
- nur mit Hilfe des BS zu Realisieren
- Standardwerkzeug fast alles BS
- Erfordern diszipliniertes Vorgehen

3.4.2 Mutexe

Metexe (Mutual Exclusion = wechselseitiger Ausschluss) ist spezielles Semaphor mit Werten 0 oder 1 (*binäres* Semaphor). „Normale“ Semaphoren nennt man auch *zählende* Semaphore. Mutexe nur für Durchsetzung wechselseitiger Ausschlüsse.

Sind leichter zum Implementieren als Semaphore. Nicht zwingend Unterstützung des BS, so dass sie z.B. in Threadpaketen verwendbar werden.

3.5 Synchronisationsprobleme

Die folgenden Standard-Synchprobleme werden mit Semaphoren gelöst.

3.5.1 Erzeuger-Verbraucher-Problem

Zwei Prozesse, producer und consumer, arbeiten zusammen. Producer erzeugt Datenobjekte die er in seiner gemeinsam genutzten Datenstruktur ablegt. Consumer entnimmt diese und verarbeitet diese weiter. Um Konsistenzprobleme zu verhindern muss bei Zugriff auf Datenstrukturen (z.B. verkettete Liste) wechselseitiger Ausschluss durchgesetzt werden. Zudem soll Consumer nur aktiv sein, wenn Datenobjekte bereitstehen.

Folgende Lösung nutzt 2 Semaphore: count hat initialen Wert 0, mutex 1:

```
semaphore count;
semaphore mutex;
producer{
while (1) {
    item i=produce();
    down (mutex);
    append(i);
    up(mutex);
    up(count); }
}
consumer{
while (1) {
    down (count);
    down (mutex);
    item i=remove();
    up(mutex);
    consume(i); }
}
```

Die Funktionen `append` fügen neues Element vom Typ `item` in die gemeinsame Datenstruktur ein, `remove` entfernt ein Element daraus.

3.5.2 Leser-Schreiber-Problem

2 Prozesse greifen auf Datenbereich zu:

- `writer` erzeugt neues Element und legt es ab
- `reader` entnimmt und verarbeitet die Objekte

Neu dabei ist, dass beide Klassen mehrere Prozesse auftreten können. Für gemeinsamen Datenbereich gelten folgende Bedingungen:

- während Prozess vom Typ `writer` auf Datenbereich zugreift, darf kein anderer Prozess darauf zugreifen
- Prozesse vom Typ `reader` dürfen gleichzeitig auf Datenbereich zugreifen (da sie Datenbestand nicht verändern!)

Verschiedene Probleme und Lösungen bei verschiedenen Prioritäten:

- Hat `reader` höhere Priorität folgt erstes Leser-Schreiber-Problem. Schreiber sollen Zugriff auf Daten nur erhalten, wenn kein Leser zugreifen möchte => `writer` könnten verhungern. Lösung nutzt 2 Semaphore: `mutex` und `write_lock` initialisiert mit Wert 1. Gemeinsam genutzte Variable `readcount` hat Wert 0 zu Beginn.
Problem: wenn `writer` fertig muss nicht zwingend `reader`, sondern kann auch neuer `writer` aktiviert werden. Dies entspricht nicht den Forderungen!

Erstes Leser-Schreiber Problem:

```
int readcount = 0;
semaphore mutex;
semaphore write_lock;
```

```
writer{
  down(writer_lock);
  write();
  up(write_lock);
}

reader{
  down(mutex);
  readcount++;
  if(readcount==1){
    down(write_lock);
  }
  read();
  down(mutex);
  readcount--;
  if (readcount==0){
    up(write_lock);
  }
  up (mutex);
}
```

Beim zweiten Leser-Schreiber-Problem sollen die Schreiber eine höhere Priorität als die Leser haben. D.h., dass die `writer` vorrangig vor dem `reader` behandelt werden. Auch hier wieder Problem des Verhungerns, diesmal für `reader`.

Zweites Leser-Schreiber-Problem:

```
int readcount, writecount;           //beide initial 0
semaphor mutex1, mutex2, mutex3, r, w; //alle initial 1
reader{                               write{
  down(mutex3);                       down (mutex2);
  down(r);                             writecount++;
  down(mutex1);                       if (writecount==1) {down (r);}
  readcount++;                         up (mutex2);
  if (readcount==1) { down(w);}       down(w);
  up (mutex1);                         write();
  up (r);                               .. up (w);
  up (mutex3);                         down (mutex2);
  read();                              writecount--;
  down(mutex1);                       if (writecount==0) {up (r);}
  readcount--;                         up (mutex2);
  if (readcount==0) {up (w);}         }
  up (mutex1);
}
```

4 Verklemmung

Beispiel: Die Prozesse wollen beide die zwei Semaphore sem1 und sem2 exklusiv belegen:

```
Process1() {                               process2() {
Down(sem1);                                down(sem2);
Down(sem2);                                down(sem1);
Do_something();                            do_some_other_thing();
Up(sem2);                                  up(sem1);
Up(sem1);                                  up(sem2);
}                                           }
```

Belegt dabei zunächst process1 sem1 und danach process2 sem2, so kann keiner der Prozesse weiterarbeiten. Jeder ist blockiert und wartet auf den anderen. Dies nennt man **Verklemmung (Deadlock)**.

4.1 Betriebsmittel

Häufig treten Verklemmungen im Zusammenhang mit exklusiver Nutzung von **Betriebsmitteln** auf.

Beispiele für BM:

- Hardware-BM: alle Geräte, z.B. Drucker, Laufwerke, ...
- Software-BM: Einträge in Systemtabellen, Datensätze, Semaphore,

Dabei wird unterschieden zwischen:

- exklusive BM: BM, die zu einem Zeitpunkt von genau einem Prozess genutzt werden können (nichtverdrängbare BM)
 - nichtexklusive BM: BM, die von mehreren Prozessen parallel genutzt werden können
- BM häufig in Klassen gleichartiger BM unterteilt, wobei Nutzung eines BM aus der Klasse exklusiv ist.

Besonders Nutzung von exklusiven BM ist verklemmungsgefährdet.

Nutzung eines EBM erfolgt in drei Schritten:

1. Belegung des BM
2. Nutzen des BM
3. Freigeben des BM

Ist ein BM beim Belegen nicht verfügbar, muss der belegendes Prozess darauf warten. Gängig dabei ist dass das BS den Prozess solange blockiert, bis BM verfügbar ist. BM lassen sich leicht durch **binäre Semaphore** modellieren.

4.2 Überblick

Tanenbaum beschreibt Verklemmung formal wie folgt:

“A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.”

4.2.1 Bedingungen

Damit Verklemmungen auftreten können, müssen folgende vier Bedingungen erfüllt sein:

1. Wechselseitiger Ausschluss (Mutual exclusion condition): Das BM ist exklusiv.
2. Behalten und Fordern (Hold and wait condition): Prozesse die BM belegen können weitere anfordern.
3. Nicht entziehbar (Non preemption condition): BM können Prozess nicht entzogen werden.
4. Zirkuläres Warten (Circular wait condition): Es gibt geschlossene Kette von Prozessen, so dass jeder Prozess auf ein BM wartet, das vom nächsten gehalten wird.

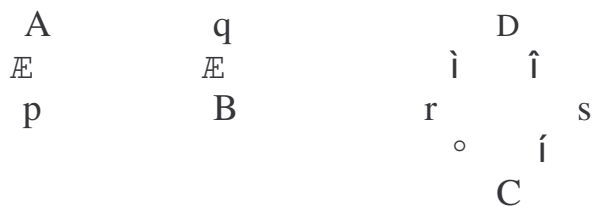
4.2.2 Belegungs-Anforderungs-Graph

Belegung von und das Warten auf BM durch Prozesse lassen sich mit Hilfe *gerichteter Graphen* beschreiben. Die Menge der Knoten E ist dabei Vereinigung der disjunkten Mengen P (der Prozesse) und R (der Betriebsmittel). Kanten können nur von Prozessen auf BM oder von BM auf Prozesse zeigen. Bedeutung dabei folgende:

- Kante von Prozess P auf BM b: P fordert b an
- Kante von BM b auf Prozess P: b ist durch P belegt

Kanten von Prozessen auf Prozesse oder von BM auf BM sind nicht erlaubt!

Verklemmung tritt auf wenn sich im Graph ein Zyklus befindet.



Die Teilgraphen haben folgende Bedeutungen:

- BM p ist durch Prozess A belegt
- Prozess B fordert BM q an
- Prozesse C und D sind über BM r und s verklemmt

P.S.: Im Skript von Herrn Friedmann werden BM in einem Viereck und Prozesse in einem Kreis dargestellt! Bitte noch mal in seinem Skript nachschauen!

4.2.3 Behandlung von Verklemmungen

Zur Behandlung von Verklemmungen gibt es vier grundsätzliche Vorgehensweisen:

- **Erkennen und Beheben:** System prüft auf Verklemmungen. Falls Verklemmung auftritt wird versucht diese zu beheben.
- **Verhindern:** System verhindert Auftreten einer der vier notwendigen Bedingungen für Verklemmungen (siehe 4.2.1 Bedingungen)
- **Vermeiden:** System prüft vor jeder Zuteilung eines BM, ob sich daraus Verklemmung ergeben könnte und verzögert ggf. Zuteilung.
- **Ignorieren:** auch eine elegante Lösung ☺

4.3 Erkennen und Beheben von Verklemmungen

(engl.: detection and recovery)

Abhängig von Anzahl der BM pro Klasse existieren verschiedene Verfahren um Verklemmungen zu erkennen.

4.3.1 Erkennung bei einem BM pro Klasse

Bei nur einem BM pro Klasse kann mittels Tiefensuche im Belegungs-Anforderungs-Graphen geprüft werden, ob dieser Zyklen enthält.

Der folgende Algorithmus untersucht jeden Knoten des Graphen getrennt, ob er die Wurzel eines Baumes ist, oder ob er sich in Zyklus befindet.

Tiefensuche zum Finden von Zyklen (vgl. Tanenbaum):

1. Wiederhole für jeden Knoten Schritt 2-6
2. L ist leere Liste, alle Kanten des Graphen seinen unmarkiert
3. Füge aktuellen Knoten N der Liste L hinzu und prüfe ob L N nun zweimal enthält. Wenn ja enthält Graph Zyklus, der in L abgelesen werden kann und Algorithmus terminiert
4. Gehen von N unmarkierte Kanten aus geht Algorithmus bei 5 weiter, sonst bei 6

Erstellt von Oliver Fleck

Fach: Betriebssysteme

Dozent: Herr M. Friedmann

Zusammenfassung Semester III

5. Wähle eine noch unmarkierte Kante und markiere sie. Folge Kante zum neuen aktuellen Knoten N und fahre bei 3. fort
6. N nicht Teil eines Zyklus. Entferne aus L. Ist L leer, so enthält Pfad keinen Zyklus. Ansonsten vorherigen Knoten zum aktuellen Knoten machen und bei 3. fortfahren

4.3.2 Erkennung bei mehreren BM pro Klasse

Bevor System mit mehreren BM pro Klasse auf Verklemmung untersucht werden kann muss es beschrieben werden. Für System mit m BM-Klassen und n Prozessen geschieht dies so:

- m BM-Klassen K_1, \dots, K_m mit E_1, \dots, E_m Betriebsmitteln
- n Prozesse p_1, \dots, p_n
- m-stelliger BM-Vektor V mit $V(j)=e_j$ (Anzahl der BM je Klasse)
- m-stelliger BM-Vektor R: freie BM je Klasse
- n x m-Belegungsmatrix B: $B(i,j)$ = von p_i belegten BM der Klasse K_j
- n x m-Belegungsmatrix A: $A(i,j)$ = von p_i geforderte BM der Klasse K_j

Es gilt die Invariante:

$$\sum_{j=1}^n B(i,j) + R(j) = V(j)$$

Um zu prüfen ob Verklemmung besteht wird geprüft ob es Ausführungsfolge gibt unter der die Forderungen aller Prozesse erfüllt werden können.

Dabei wird angenommen dass jeder Prozess nach Forderung erfüllt wurde terminiert wird und belegte Ressourcen wieder freigibt.

Geschieht dies nicht ist Prozess verklemmt. Andere Prozesse können so durch weitere Forderungen ebenfalls in Verklemmungssituation kommen.

Verfahren findet nur verklemmte Prozesse, nicht aber solche die verklemmen könnten!

1. alle Prozesse unmarkiert, für m-stelligen Vektor R' gilt $R'(i) = R(i)$; $i=1..m$
2. suche unmarkierten Prozess p_i der Bedingung $\dots i \in \{1..n\} : A(i,j) < R'(i)$ erfüllt.
Anforderungen dieses Prozesses können erfüllt werden
3. falls solcher Prozess existiert gib seine BM (scheinbar) frei: $R'(j) = R'(j) + B(i,j)$ für $j \in \{1..m\}$
4. falls solcher Prozess existiert terminiert das Verfahren. Alle nicht markierten Prozesse sind verklemmt

4.3.3 Wann wird Erkennung ausgeführt

Vorgestellte Verfahren zur Erkennung von Verklemmungen sind rechenaufwändig. Deshalb ist zu klären, zu welchem Zeitpunkt sie eingesetzt werden. Möglichkeiten sind:

- immer wenn BM-Anforderung nicht sofort erfüllt werden \checkmark kann sehr aufwendig
- wenn CPU-Auslastung unter bestimmte Schwelle sinkt \checkmark solange CPU-lastige Prozesse aktiv sind, werden keine Verklemmungen festgestellt
- abhängig von Verklemmungs-Wahrscheinlichkeit in regelmäßigen Intervallen

4.3.4 Behebung von Verklemmungen

Wurde Verklemmung erkannt gibt es mehrere Möglichkeiten zur Beseitigung:

- (manuelles) Unterbrechen (Preemption) eines Prozesses
- teilweise Wiederholung (Rollback) eines Prozesses
- Abbrechen eines Prozesses

Alle Möglichkeiten haben spezielle Nachteile...

- Behebung durch Unterbrechen

Seite 26 von 45

Erstelldatum 12.11.2002 1:51

Erstellt von Oliver Fleck

Fach: Betriebssysteme

Dozent: Herr M. Friedmann

Zusammenfassung Semester III

- verklemmter Prozess wird unterbrochen
- BM entzogen
- i.A. manueller Eingriff nötig

- Beheben durch Wiederholen

- während Ablauf Checkpoints angelegt für Prozesszustand und Zustand aller BM
- zur Verklemmungsvermeidung wird Prozess an früheren Checkpoint zurückgesetzt um BM freizugeben
- alle nach dem Checkpoint verrichtete Arbeit geht verloren
- Aufwendige Speicherung nötig

- Beheben durch Abbrechen

- ein oder mehrere Prozesse werden abgebrochen um Verklemmung zu lösen
- Problem: was ist mit Änderungen, die Prozess bereits ausgeführt hat (z.B. Datenbank)

- Bewertung

Keines der Verfahren ist sonderlich attraktiv. Alle haben folgende Probleme:

- an welchem Prozess soll angesetzt werden?
- wie wird Verhungern von Prozessen verhindert?
- keines ist universell einsetzbar!

4.4 Verhindern von Verklemmung

Um Verklemmung zu verhindern (Deadlock Prevention) muss eine der vier notwendigen Bedingungen gebrochen werden. Im folgenden wird für jede dieser Bedingungen gezeigt, ob und wie das möglich ist...

4.4.1 Wechselseitiger Ausschluss

Für viele BM ist wechselseitiger Ausschluss nötig. Ansatz Spooling:

- nur ein spezieller Prozess darf auf BM direkt zugreifen
- dieser Prozess sammelt E/A-Aufträge der andere Prozesse und speichert diese auf Platte zwischen
- Problem: knapper Plattenplatz kann zu Verklemmung führen
- Problem: Nicht alle BM lassen sich so behandeln

Generelle Idee zum reduzieren der Verklemmungsgefahr:

- möglichst wenige Prozesse sollen BM belegen
- BM sollen möglichst selten belegt werden
- ◊ für manche BM einsetzbar, aber nicht für alle

4.4.2 Behalten und Fordern

- Prozesse müssen alle BM am Start belegen
- Problem: manche wissen nicht was sie wann benötigen
- Alternative: vor nachfordern aller BM freigeben, dann alle gemeinsam wieder anfordern
- ineffiziente Nutzung der BM, da diese lange gehalten werden ohne im Einsatz zu sein

4.4.3 Nicht-Entziehbarkeit

Es gibt BM die nicht entziehbar sind. Auf diese Weise lässt sich das Problem nicht verhindern.

4.4.4 Zirkuläres Warten

Es wird Vorgehen versucht dass Zyklen im Belegungs-Anforderungs-Graphen verhindert.

Erstellt von Oliver Fleck

Fach: Betriebssysteme

Dozent: Herr M. Friedmann

Zusammenfassung Semester III

- Einfache Lösung: Prozess darf nur ein BM halten; i.A. nicht geeignet

- Alternative:

- Einführen einer eindeutigen Nummerierung der BM
- es dürfen nur BM angefordert werden deren Nummer höher ist als die höchste Nummer der bereits durch den Prozess belegten BM
- Problem: Finden einer Ordnung die für alle Anwendungen sinnvoll ist

4.4.5 Bewertung

Verfahren sind, wenn überhaupt, nur in Spezialfällen einsetzbar.

4.5 Vermeidung

Hier wird Verfahren vorgestellt, das bei dynamischen Anforderungen von BM das Auftreten von Verklemmungen vermeidet. Voraussetzung hierfür ist, dass für jeden Prozess die maximale Anzahl zu belegender BM pro Klasse bekannt ist. So kann geprüft werden ob es nach der Belegung eines BM noch möglich ist eine Zuteilungsreihenfolge für BM-Forderungen aller Prozesse zu finden, so dass alle Prozesse ausgeführt werden können.

4.5.1 Beschreibung des Systems

Eine Menge von Prozessen und BM wird wie folgt beschrieben:

- m BM-Klassen K_1, \dots, K_n mit E_1, \dots, E_n BM
- n Prozesse p_1, \dots, p_n
- m-stelliger BM-Vektor V mit $V(j) = E_j$ (Anzahl der BM je Klasse)
- m-stelliger BM-Vektor R : freie BM je Klasse
- n x m-Belegungsmatrix B : $B(i,j)$ = von p_i belegte BM der Klasse K_j
- n x m-Anforderungsmatrix A : $A(i,j)$ = maximale Restforderung von p_i für Klasse K_j

4.5.2 Sichere und unsichere Zustände

Zustand des Systems als sicher bezeichnet wenn es eine Zuteilungsfolge für die BM gibt, so dass alle Prozesse ohne Verklemmung ausgeführt werden können.

Unsicher gilt das System, wenn es keine Zuteilungsfolge gibt, aber auch nicht verklemmt ist.

Ein unsicherer Zustand muss also nicht zur Verklemmung führen, da jedoch nichts über die Zeitpunkte der Freigabe bekannt ist, kann nicht festgestellt werden, ob der Zustand tatsächlich zu einer Verklemmung führt.

4.5.3 Bankier's Algorithmus

Dieser Algorithmus vermeidet Verklemmungen wie folgt:

1. wird in einem Zustand (A,B,R) ein BM angefordert, berechne (ohne das BM zu belegen) den Folgezustand (A',B',R')
2. prüfe ob (A',B',R') ein sicherer Zustand ist
3. ist (A',B',R') sicher belege das BM

Bleibt noch zu klären, wie geprüft wird, ob ein Zustand sicher ist:

Ein Zustand ist sicher, wenn Ausführungsreihenfolge der Prozesse vorhanden ist, so dass jeder Prozess seine Forderungen erfüllt bekommt. Dies ist der Fall wenn das Verfahren zum Erkennen von Verklemmungen aus 4.3.2 für (A',B',R') keine Verklemmungen findet.

Verfahren nimmt an, dass alle Prozesse gesamte benötigte BM auf einmal anfordert ohne zwischendurch andere BM freizugeben. Da dies nicht zwangsläufig auftreten muss kann es sein dass auch unsichere Zustände nicht zu Verklemmungen führen.

Erstellt von Oliver Fleck

Fach: Betriebssysteme

Dozent: Herr M. Friedmann

Zusammenfassung Semester III

Algorithmus wird manchmal als Habermann-Verfahren bezeichnet.

4.5.4 Bewertung

Bankier's Algorithmus erlaubt es eine Menge von Prozessen verklemmungsfrei auszuführen. Allerdings mit Forderung, dass die max. BM-Belegung aller Prozesse vorab bekannt ist. Diese Info häufig nicht vorhanden.

Weiterhin kann er BM nicht optimal zuteilen da keine Info darüber, welche BM schon vor Ende eines Prozesses wieder freigegeben werden. System also nicht optimal ausgelastet!

4.6 Ignorieren

- aus theoretischer Sicht total unbefriedigend!

ABER:

- kein anderes Verfahren universell einsetzbar
- drei Parameter betrachten:
 - i) Häufigkeit von Verklemmungen
 - ii) Kosten von Verklemmungen
 - iii) Kosten der anderen Verfahren

Gängige BS, sowohl UNIX als auch Windows ignorieren Verklemmungen und lassen sie, wenn sie auftreten von Benutzer behandeln.

Das heißt aber nicht, dass es nicht Anwendungen gibt, die andere Strategien einsetzen, z.B. verwenden Datenbanksysteme für die von ihnen bereitgestellten BM (z.B. Einträge, Tabellen,...) spezielle Zuteilungsstrategien, um Verklemmungen zu vermeiden (oder zu verhindern...).

5 Hauptspeicherverwaltung

Hauptspeicher ist ein zentrales BM dass unter Prozessen aufgeteilt werden muss. Ziele der Verwaltung sind z.B.:

- möglichst viele Programme lauffähig halten
- Schutz der Programme voreinander
- ggf. Simulation von mehr Hauptspeicher als real Vorhanden (virtueller Speicher)
- Einfache Nutzung des Speichers

Zentraler Begriff dabei ist die **Adresse**. Es wird zwischen **physikalischen** Adressen, die einer bestimmten Stelle des Speichers entsprechen, und **logischen** (bzw. **virtuellen**) Adressen, die innerhalb der Programme verwendet werden, unterschieden. Im einfachsten Fall sind logische Adressen = physikalischen Adressen. Für viele Anwendungen ist es jedoch nützlich zwischen den Adressen andere Zuordnung (ggf. auch dynamisch) zu verwenden. Diese Umsetzung wird von der Hardware in der MMU (Memory Management Unit) vorgenommen, die in modernen Systemen meist in der CPU integriert ist.

Adressen werden zu **Adressräumen** zusammengefasst. Ein Adressraum beginnt normalerweise bei Adresse 0 und endet bei einer Obergrenze. Obergrenze des physikalischen Adressraumes durch Hauptspeicher vorgegeben. Logische Adressräume können abweichend kleiner oder größer sein.

5.1 Einfache Speicherverwaltung

Hier gibt es Verfahren zur Speicherverwaltung die den Hauptspeicher für ein oder mehrere Programme bereitstellen, ohne dass Auslagern des Speicherinhalts auf andere Speichermedien eingesetzt wird. Dabei sind vier Teilgebiete zu untersuchen:

- Adressierung des Speichers
- Schutz des Speichers
- Aufteilung des Speichers auf Prozesse
- Zuteilung von Speicherbereichen

Adressierung und Speicherschutz sind dabei typische Mechanismen, die meist in Hardware realisiert ist. Bei Auf- und Zuteilung des Hauptspeichers lassen sich verschiedene Strategien betrachten.

5.1.1 Adressierung

Zwei Einfache Arten der Adressierung des Hauptspeichers: **direkte** und **relative** Adressierung.

Diese Verfahren unterscheiden sich darin, wie die (logischen) Adressen innerhalb der Programme auf (physikalische) Adressen des Speichers abgebildet werden.

Weitere kompliziertere Adressierungsmechanismen folgen später.

5.1.1.1 Direkte Adressierung

Alle logischen Adressen innerhalb eines Programms (z.B. bei Sprüngen oder Speicherzugriffen) als die physikalischen Speicheradressen interpretiert. Hierbei verschiedene Probleme:

- schlechte Verschiebbarkeit der Programme im Hauptspeicher:
wird Programm im Speicher verschoben müssen alle Bezüge auf Adressen innerhalb des Programms angepasst werden

Erstellt von Oliver Fleck
Fach: Betriebssysteme
Dozent: Herr M. Friedmann
Zusammenfassung Semester III

- Programmgröße:
sind Programme größer als der verfügbare Hauptspeicher müssen umständliche Techniken (z.B. Overlays) eingesetzt werden, um Programm in Module aufzuteilen, die unabhängig voneinander ausgeführt werden können

5.1.1.2 Relative Adressierung

Im Einprozess-Betrieb ist es noch denkbar, Systeme zu erstellen, die alle Programme immer an feste Stelle im Speicher laden, so dass vermieden wird, die Programmadressen bei jedem Laden neu anzupassen. Im Mehrprozess-Betrieb ist dies ausgeschlossen.

Mit relativen Adressierung wird Anpassen der Programmadressen geschickt vermieden: Alle Programmadressen werden relativ zu einer festen Basisadresse betrachtet. Bei jedem Speicherzugriff wird der Wert der Basisadresse zur logischen Adresse addiert, um die physikalische Adresse zu ermitteln. Die Programme können so geschrieben werden, als ob sie sich ab Adresse 0 im Speicher befinden. Beim Laden bzw. Verschieben der Programme im Speicher muss dann nur Basisadresse entsprechend angepasst werden.

Addition erfolgt üblicherweise der von der Hardware ausgeführt. Dazu wird Basisadresse in speziellem Register der MMU abgelegt. Beim Multitasking muss Register bei jedem Prozesswechsel entsprechend ersetzt werden.

5.1.2 Schutz des Speichers

- Ziel: Jedes Programm darf nur auf zugewiesenen Speicherbereich zugreifen
- Speicherbereich ist zusammenhängende Folge von Adressen fester Länge
- bei jedem Zugriff Prüfung ob Bereich verletzt wird:
 - bei direkte Adressierung: Vergleich mit Ober- und Untergrenze
 - bei relativer Adressierung: logische Adresse muss größer oder gleich 0 und kleiner als Bereichslänge sein
- bei Bereichsverletzung: Trap ins BS, dort Fehlerbehandlung
- Bereichsprüfung normalerweise in MMU (Hardware!)
- bei Prozesswechsel Anpassen der entsprechenden Register in MMU

5.1.3 Aufteilung des Hauptspeichers

5.1.3.1 Aufteilung bei Einprozessor-Systemen

Im Falle von Einprozess-Systemen Aufteilung des Speichers sehr einfach. BS in einem Teil des Speichers der am Rande des verfügbaren Adressraumes gelegen ist. Rest des Speichers ist als zusammenhängender Block für das gerade ausgeführte Programm verfügbar.

5.1.3.2 Aufteilung bei Mehrprozessor-Systemen

Bei Mehrprozess-Systemen steht jedem Prozess ein zusammenhängender Abschnitt des Speichers zur Verfügung. Dabei gibt es zwei wesentliche Strategien den Speicher aufzuteilen: Bereiche mit fester Größe und dynamische Aufteilung.

Bereiche fester Größe:

- Anzahl und Größe der Bereiche bei Systemstart festgelegt
- Zuteilungsstrategien der Aufträge an die Bereiche:
 - eine Warteschlange pro Speicherbereich
 - gemeinsame Warteschlange

Erstellt von Oliver Fleck
Fach: Betriebssysteme
Dozent: Herr M. Friedmann
Zusammenfassung Semester III
- dynamische Zuteilung:

- für jedes Programm wird beim Laden ein passender Speicherbereich belegt
- Speicherbereiche haben beliebige Größe
- Probleme: Zuteilungsstrategie, Fragmentierung

5.1.4 Zuteilung

Zwei Grundlegende Funktionen bei dynamischen Einteilung: **Belegen** und **Freigeben**
Durch diese Funktionen wird zunächst komplett freier Speicher nach und nach in eine Folge von freien und belegten Abschnitten aufgeteilt. Ziel der Aufteilung (d.h. beim Belegen) ist es den „Verschnitt“ zu minimieren, d.h. möglichst wenige sehr kleine Speicherabschnitte entstehen zu lassen, die man später schlecht nutzen kann. Neben der Zuteilung ist eine wichtige Frage die Datenstruktur, mit der die Belegung des Speichers festgehalten wird. Die hier vorgestellten Verfahren werden auch zur programminternen Verwaltung des Freispeichers (Heap) verwendet, die unabhängig vom BS ist.

5.1.4.1 Datenstrukturen

Zwei grundsätzliche Möglichkeiten: Bitmaps und verkettete Listen

Bitmaps:

- Speicher in kleine Einheiten fester Größe unterteilt
- in Bitmap wird für jede Einheit festgehalten ob sie frei oder belegt ist
- beim Freigeben werden betroffene Einheiten in Bitmap als frei gekennzeichnet
- beim Belegen muss zusammenhängende Folge gefunden werden deren Gesamtgröße genügt
- Problem: Größe der zusammenhängenden Bereiche nur implizit in Bitmap abgelegt → aufwendige Suche

verkettete Listen:

- Speicher als Folge zusammenhängender freier und belegten Bereiche betrachtet, wobei für jeden Bereich die Länge explizit bekannt ist
- Belegung von Speicherbereichen:
 - Suche eines freien Bereichs, der mindestens so groß ist wie der angeforderte Bereich
 - wenn gefundener Bereich zu groß ist: Aufteilen:
 - neu belegter Bereich; bei manchen Systemen nicht genaue Größe, sondern Vielfaches einer Minimalgröße
 - Rest: neuer Freibereich; wird, wenn vorhanden, mit angrenzendem Freibereich verbunden
 - wenn Rest sehr klein wird er bei manchen Systemen nicht abgegrenzt sondern der gesamte Bereich belegt
- beim Freigeben werden sich berührende freie Bereiche zusammengefügt
- Bereiche werden in Listen verwaltet, Möglichkeiten:
 - gemeinsame Liste für freie und belegte Bereiche:
 - leichtes Zusammenfügen von freien Bereichen, langsames Durchsuchen
 - getrennte Listen für frei und belegt:
 - schnelles durchsuchen, Zusammenfügen schwieriger
 - verschiedene Listen für verschiedene große freie Bereiche:
 - schnelles Auffinden eines besonders gut passenden Bereiches, Zusammenfügen noch schwieriger

5.1.4.2 Strategien

Zum Suchen eines freien Speicherbereiches gibt es verschiedene Strategien mit spezifischen Vor- und Nachteilen. Im folgenden wird von verketteten Listen für die freien Bereiche ausgegangen, bei Bitmaps entsteht zusätzlicher Aufwand zur Bestimmung der Größe der Bereiche.

First-Fit:

- Liste wird vom Anfang durchsucht, der erste passende freie Bereich wird verwendet

Next-Fit/Rotating-First-Fit:

- Suche wird nach dem letzten gefundenen Freibereich oder dem letzten freigegebenen Bereich fortgesetzt

Best-Fit:

- Liste wird komplett durchsucht, derjenige Bereich, bei dem der geringste Rest übrig bleibt wird verwendet
- Vorteil: große Freibereiche bleiben erhalten
- Nachteil: Restbereiche sehr klein und schwer zu nutzen

Worst-Fit:

- wie Best-Fit, aber Bereich mit dem größten Rest wird verwendet
- Vorteil: Rest kann noch gut weiterverwendet werden
- Nachteil: große Freibereiche werden rasch aufgeteilt

Buddy-Verfahren:

Die bisherigen Verfahren belegen Speicherbereiche beliebiger Größe und fügen diese nach dem Freigeben wieder zusammen. Anders das Buddy-Verfahren:

Nur Speicherblöcke, deren Größe eine Zweierpotenz (2^n) ist, festgelegt. Zudem Mindestgröße festgelegt. Für jede zulässige Größe wird eigene Liste freier Blöcke erstellt.

Wird Block angefordert, dessen Größe die Bedingungen nicht erfüllt, wird Größe zur nächsten zulässigen Größe erhöht.

Ist kein Block der geforderten Größe frei, so wird rekursiv Block der nächsten Größe angefordert und in zwei Blöcke, die sog. Buddies („Kumpel), aufgeteilt. Einer der Buddies wird in Freiliste der geforderten Größe eingetragen, der andere an den Aufrufer weitergegeben. Das rekursive Anfordern wird solange fortgesetzt, bis freier Block gefunden wurde, oder (auf der höchsten Ebene) festgestellt wird, dass kein passender Block verfügbar ist.

Beim Freigeben wird geprüft ob sein Buddy ebenfalls frei ist. In diesem Fall wird der Buddy aus der Freiliste entfernt, die beiden Buddies zu einem Block der nächsten Zweierpotenz zusammengefügt und als solcher freigegeben. Wie beim Belegen wird auch Freigabe rekursiv nach oben fortgesetzt.

Buddy-Verfahren findet sehr schnell passenden Speicherbereich, da es in jeder Freiliste maximal einmal nachschauen muss. Zudem wird vermieden, dass große Speicherblöcke aufgeteilt werden, solange kleinere Vorhanden sind.

5.1.4.3 Fragmentierung und Kompaktierung

Durch wiederholtes Anfordern und Freigeben zerfällt Speicher nach und nach in Folge von freien und belegten Abschnitten. Dadurch wird es zunehmend schwieriger große zusammenhängende Bereiche zu belegen. Diese Zerstückelung nennt man **externe Fragmentierung**. Je nach Belegungsstrategie können dabei erhebliche Mengen des Hauptspeichers verschwendet werden.

Erstellt von Oliver Fleck
Fach: Betriebssysteme
Dozent: Herr M. Friedmann
Zusammenfassung Semester III

Unter interner Fragmentierung versteht man die Zuteilung eines größeren Speicherbereiches als angefordert wurde. Der dabei zu viel zugeteilte Speicher ist nach Freigabe des gesamten Bereiches wieder allgemein nutzbar und liegt solange brach.

Durch Kompaktierung können bei externe Fragmentierung die freien Speicherbereiche zu einem größeren Bereich zusammengefasst werden. Dazu werden belegte Bereiche zum Rand des physikalischen Adressraumes verschoben, so dass zwischen ihnen keine Lücken mehr sind. Dies hat zur Folge, dass freie Bereiche automatisch am anderen Rand gebracht werden, wo sie zu einem großen Freibereich verschmelzen können.

Kompaktierung erfordert Verschiebbarkeit der Programme. Bei relativer Adressierung erfolgt dies automatisch, ansonsten müssen alle Adressbezüge in den Programmen angepasst werden.

5.2 Swapping

Bisherige Verfahren erfordern dass alle Prozesse ständig im Speicher gehalten werden. Dies stellt erhebliche Einschränkung dar, da blockierte Prozesse Speicher belegen. Um die Auslastung zu steigern könnte dieser Speicher von Programmen genutzt werden, die rechenbereit sind.

Einfacher Ansatz zur Lösung des Problems: **Swapping**.

Swapping lagert blockierte Prozesse auf **Hintergrundspeicher** aus (i.A. Festplatte). Wird ausgelagerter Prozess wieder rechenbereit muss er lediglich wieder in Hauptspeicher geschoben werden. Die strategische Entscheidung über Ein- und Auslagern erfolgt meist von einem speziellen Scheduler (vgl. 2.2.2: Three-Level-Scheduling).

Das Ein- und Auslagern erfolgt immer mit kompletten Prozessen. Es muss beachtet werden, dass Prozesse, deren Ein- bzw. Ausgabeoperationen über DMA abgewickelt werden, nicht ausgelagert werden dürfen!

5.3 virtueller Speicher

Bisherige Verwaltung des Hauptspeichers haben zwei wesentliche Nachteile:

1. maximale Größe der Blöcke durch Größe des Hauptspeichers beschränkt
2. Speicherbereiche müssen zusammenhängend sein, was die die Belegung zusätzlich erschwert und häufig zur Fragmentierung des Hauptspeichers führt

Beide Probleme werden elegant durch Verwendung von **virtuellem Speicher** gelöst.

Zentraler Mechanismus dabei ist das sog. **Paging**. Ergänzt wird das Paging durch die sog. Seitenersetzungsstrategie.

5.3.1 Paging

Beim Paging werden alle Adressen mit denen ein Prozess operiert als virtuelle Adressen bezeichnet. Sie bilden den virtuellen Adressraum des Prozesses. Der virt. Adressraum ist zusammenhängende Folge von Adressen. Weiterhin virt. Adressraum eine Folge von **Seiten**. Größe dieser Seiten ist eine Zweierpotenz. Hauptspeicher wird als Folge von **Seitenrahmen (page-frames)** der gleichen Größe betrachtet.

Die MMU setzt virt. Adressen in physikalische um. Dabei nimmt sie eine Zuordnung der Seiten zu den Seitenrahmen vor. Jede Seite kann einem beliebigen Seitenrahmen zugeordnet werden d.h. virtuell zusammenhängende Adressräume bilden im physikalischen Speicher keinen zusammenhängenden Block mehr).

Aufruf einer nicht im Hauptspeicher präseneter Seite führt zu Seitenfehler (**page-fault**). Dabei wird via Trap ins BS verzweigt und der laufende Prozess blockiert. Das BS lagert die Seite in einen freien Rahmen ein und setzt danach den Prozess fort. Ist keiner der Frames frei, muss der Inhalt eines belegten Frames ausgelagert werden, um Platz zu schaffen.

Erstellt von Oliver Fleck

Fach: Betriebssysteme

Dozent: Herr M. Friedmann

Zusammenfassung Semester III

Hintergrundspeicher für Seitenrahmen in der Regel Teil der Festplatte. Hierbei kommen häufig spezielle Dateisysteme zum Einsatz, die schnelles Ein- und Auslagern ermöglichen.

Durch Ein- und Auslagern können die einzelnen virt. Adressräume größer als die physik. vorhandenen Speicher sein.

Da Seitenrahmen beim virt. Adressraum nicht zusammenhängend im Speicher abgelegt werden müssen verschwindet auch Problem der Fragmentierung.

Solange kein Ein- bzw. Auslagern vorgenommen wird, wird Paging autonom von der Hardware erledigt. Erst bei einem Seitenfehler muss BS eingreifen.

5.3.1.1 virtuelle Adressen

Eine virt. Adresse zerfällt in zwei Teile. Bei Seitengröße von s^n Byte werden niederwertige n Bits der Adresse zum Ansprechen eines der Bytes der Seite verwendet, die höherwertigen Bits bestimmen die Nummer der Seite. Über die Nummer wird die **Seitentabelle (page-table)** angesprochen, in der die Zuordnung von (virtuelle) Seiten und (physikalischen) Seitenrahmen (üblicherweise im Hauptspeicher) abgelegt ist.

Die virt. Adresse lässt sich als Tupel (p,d) betrachten, wobei p die Nummer der Seite und d die Position innerhalb der Seite ist. Berechnung der physik. Adresse aus der virt. Adresse:

Physik. Adresse = Seitenrahmenanfang $(p) + d$

Diese Adressberechnung (incl. Zugriff auf Seitentabelle) wird in Hardware durch MMU vorgenommen!

5.3.1.2 Seitentabelle

Für jeden virt. Adressraum wird eigene Seitentabelle unterhalten. Tabelle speichert für jede Seite des Adressraumes typischerweise folgende Informationen:

- Present-Bits: Status der Seite (ein- oder Ausgelagert)
- Page-Frame: Nummer des zugeordnete Seitenrahmens
- Modified-Bit: Seite wurde verändert (muss beim Auslagern tatsächlich geschrieben werden)
- Referenced-Bit: auf Seite wurde zugegriffen
- Protection-Bit(s): Schutzstatus, z.B. nur-Lesen, Lesen und Schreiben, ausführbar, ...
- Caching-Disabled: z.B. nur Wahrung der Konsistenz im DMA-Betrieb

In Seitentabelle nur diejenigen Infos gespeichert, auf die die Hardware (speziell MMU) direkt zugreifen muss; für weitere Infos, z.B. den der Seite zugeordneten Platz auf den Hintergrundspeicher, verwaltet BS gesonderte Tabellen.

Tabellen können sehr groß werden. Bei 4kB Seitenrahmen und 32-bit virt. Adressen gibt es z.B. $\approx 1.000.000$ Seiten. Häufig werden deshalb zwei- oder mehrstufige Tabellen verwendet.

Bei einem zweistufigen System wird die virt. Adresse als Tripel (p,q,d) betrachtet, wobei p aus einer übergeordneten Tabelle eine Seitentabelle auswählt. Mit q wird dann eine Seite ausgewählt innerhalb der mit d adressiert wird. Mehrstufige Verfahren interteilen Adresse in entsprechend mehr Bestandteile und suchen die Seite über mehrere Tabellen. Üblich sind aber nur zwei- und dreistufige Verfahren.

Durch Verwendung mehrstufiger Tabellen lässt sich Speicherplatz einsparen, da nur für tatsächlich benötigten Seiten des virt. Adressraumes Seitentabellen angelegt werden müssen.

Zudem ist übergeordnete Tabelle nötig, um richtige Seitentabelle zu finden.

Bei mehrstufigen Systemen erhöht sich Aufwand für Speicherzugriffe, da MMU einen zusätzlichen Zwischenschritt durchführen muss.

Umsetzung virt. Adresse – physik. Adresse durch MMU wird durch dazu erforderlichen Zugriff auf Seitentabelle erheblich gebremst (pro Zugriff des Prozessors muss MMU einmal auf Speicher zugreifen!). Zur Beschleunigung werden TLBs (Transaction Lookaside Buffers)

Erstellt von Oliver Fleck

Fach: Betriebssysteme

Dozent: Herr M. Friedmann

Zusammenfassung Semester III

eingesetzt. Dies sind Zwischenspeicher, in denen Einträge der Seitentabellen für die letzten Seiten von der MMU vorgehalten werden. Wird auf Seite zugegriffen, deren Tabelleneintrag dort abgelegt ist, muss nicht auf Seitentabelle im Hauptspeicher zugegriffen werden.

Da häufig auf nahe beieinander liegende Speicherstellen zugegriffen wird (Lokalitätsprinzip), muss nur selten auf eigentliche Seitentabelle zugegriffen werden.

5.3.2 Seitenersetzungsstrategie

Wird beim Paging neue Seite eingelagert, kann der Fall eintreten, dass für diese kein Platz im physik. Hauptspeicher vorhanden ist. In diesem Fall muss eine der vorhandenen Seiten ausgelagert und die neue eingelagert werden.

Da dies erheblichen Zeitaufwand bedeutet, ist man bestrebt, möglichst selten ein- bzw. auszulagern.

Beim Einlagern wenig Wahlmöglichkeiten: Soll bestimmter Prozess weiterlaufen, der eine Seite benötigt, so muss diese eingelagert werden.

Beim Auslagern in der Regel mehrere Möglichkeiten (nämlich die Anzahl der belegten Seitenrahmen). Eine optimale Strategie wählt hier Seitenrahmen aus, der am längsten ungenutzt ist. Da System nicht in Zukunft schauen kann, ist es hier darauf angewiesen, aus dem zurückliegenden Zugriffsverfahren auf die Zukunft zu schließen.

Weiteres Kriterium ist, ob Seiten verändert wurden: Seiten veränderter Seitenrahmen müssen auf Hintergrundspeicher entsprechend überschrieben werden, während unveränderte Seitenrahmen (sofern sie schon auf Hintergrundspeicher stehen) einfach beim Einlagern der neuen Seiten überschrieben werden können.

Es folgen nun verschiedene Strategien und weitere Aspekte. Den Schwerpunkt bilden zunächst Strategien zur **Auslagerung** von Seiten. Es folgen danach Strategien, die sich mit dem **Einlagern** von Seiten befassen.

5.3.2.1 Globale und lokale Seitenersetzung

Generell lassen sich globale und lokale Seitenersetzung unterscheiden.

Bei **lokalen** Seitenersetzung verfügt jeder Prozess über feste Menge von Seitenrahmen, in die seine Seiten eingelagert werden können. Muss neue Seite eingelagert werden, wird eine andere Seite des Prozesses verdrängt.

Bei **globalen** Seitenersetzung werden alle Prozesse aus einer gemeinsamen Menge von Seitenrahmen versorgt. Hierbei können Seiten eines anderen Prozesses verdrängt werden.

Bei globaler Seitenersetzung kann Zugriffsverhalten eines Prozesses die Leistungsfähigkeit anderer Prozesse beeinflussen. Meist wird dabei Durchsatz des Gesamtsystems auf Kosten einzelner Prozesse gesteigert.

5.3.2.2 Not Recently Used (NRU)-Algorithmus

- BS untersucht Referenced- und Modified-Bit für alle Seiten
- Referenced-Bit wird regelmäßig zurückgesetzt (z.B. über Timer-Interrupt)
- Unterteilung der Seiten in 4 Klassen:
 1. Klasse 0: nicht referenziert, nicht verändert
 2. Klasse 1: nicht referenziert, verändert
 3. Klasse 2: referenziert, nicht verändert
 4. Klasse 3: referenziert, verändert
- ausgelagert wird immer eine Seite der nichtleeren Klasse mit der kleinsten Nummer

- Idee:
 - o Seiten die länger nicht referenziert wurden werden vermutlich auch nicht so schnell wieder gebraucht
 - o Auslagern von nicht-veränderten Seiten wird bevorzugt, da diese nicht geschrieben werden müssen
- Bewertung:
 - o Sehr grobe Abschätzung der Eigenschaft „lange nicht referenziert“
 - o Auch stark frequentierte Seiten können ausgelagert werden, wenn ihr Ref-Bit kurz vor Auslagerung gelöscht wurde
 - o Aufwändige Suche nach Seite zum Auslagern, da zunächst Klasseneinteilung vorgenommen werden muss

5.3.2.3 First-In, First-Out (FIFO)-Algorithmus

- BS unterhält Liste der eingelagerten Seiten
- beim Einlagern kommt Seite ans Ende der Liste
- beim Auslagern wird immer die vorderste Seite der Liste ausgelagert
- Bewertung:
 - sehr einfach zu implementieren
 - schnelle Ausführung
 - schlechte Auswahl: auch extrem häufig genutzte Seiten werden ausgelagert; alle Seiten werden gleich behandelt

FIFO-Anomalie (Belady's Anomalie):

Man sollte erwarten, dass das Erhöhen der Anzahl der Seitenrahmen die Häufigkeit der Seitenfehler reduziert. Beim FIFO-Algorithmus lassen sich jedoch Zugriffsfolgen finden, die bei Erhöhen der Seitenzahl mehr Seitenfehler erzeugen.

5.3.2.4 Second-Chance-Algorithmus

- Erweiterung des FIFO-Algorithmus
- Einlagern: Seite kommt ans Ende der Liste
- Auslagern: am Anfang der Liste, unter Beachtung des Referenced-Bits:
 - Seite nicht referenziert: auslagern
 - Seite referenziert: Ref-Bit löschen, Seite bleibt eingelagert und kommt an Anfang der Liste; Liste wird solange durchsucht, bis unreferenzierte Seite gefunden wird
- ggf. zyklisches Löschen des Referenced-Bit
- Bewertung:
 - leicht zu implementieren
 - Zeit zur Suche nach auslagerbaren Seite abhängig von Anzahl der eingelagerten Seiten
 - Seiten die regelmäßig genutzt werden, werden nicht ausgelagert

5.3.2.5 Clock-Algorithmus

- effizientere Implementierung für 2nd-Chance-Algorithmus
- Seiten werden als zyklische Liste betrachtet
- Zeiger durchwandert Liste zyklisch
- Seitenfehler: untersuche Referenced-Bit der aktuellen Seite
 - gesetzt: Lösche Bit und gehe zur nächsten Seite
 - nicht gesetzt: Seite wird ausgelagert

5.3.2.6 Clock-Algorithmus mit zwei Zeigern

- ein Zeiger zum Auslagern falls Referenced-Bit gelöscht ist
- ein Zeiger zum Löschen des Referenced-Bit
- beide Zeiger werden gemeinsam bewegt (haben festen Abstand)
- Feinabstimmung des Verfahrens durch Einstellen des Zeigerabstandes
- je größer der Abstand der Zeiger, desto größer ist die Chance, dass eine Seite wieder referenziert wird, bevor das Referenced-Bit zwecks Auslagerung untersucht wird

5.3.2.7 Least Recently Used (LRU)- Verfahren

- Überlagerung:
 - Seiten, die in der nahen Vergangenheit häufig genutzt wurden, werden vermutlich weiterhin stark genutzt
 - Seiten die lange ungenutzt wurden, werden wohl in Zukunft ebenfalls kaum genutzt
- Verfahren: beim Auslagern wird immer Seite gewählt, die am längsten ungenutzt war
- Algorithmus ist realisierbar, aber extrem aufwändig!
- Lösung in Hardware:
 - Zeitmarke für jede Seite in Tabelle; Setzen nach jedem Zugriff durch MMU
 - nxn-Bitmatrix in Hardware (bei n Seitenrahmen):
 - bei Zugriff auf Rahmen n: setze alle Bits in Zeile n auf 1, danach alle Bits in Spalte n auf 0
 - binärer Wert der Zeilen gibt an, welcher Rahmen wie lange nicht genutzt wurde: je kleiner Wert, desto länger ungenutzt
- gesucht werden gute Approximationen in Software!

5.3.2.8 Not Frequently Used (NFU)- Verfahren

- Simulation von LRU in Software
- Idee: genauer abschätzen, wie oft auf Seite zugegriffen wird
- Vorgehen:
 - Zähler für jede Seite; Anfangswert 0
 - regelmäßig (z.B. Zeitgeber-Interrupt) wird für jede Seite der Wert des Ref-Bits zum Zähler addiert und anschließend das Bit gelöscht
 - zum Auslagern wird Seite mit dem geringsten Zählerstand gewählt
- Problem: Verfahren vergisst nicht: hohe Zählerstände bleiben erhalten; Seiten die in Vergangenheit intensiv genutzt wurden bleiben im System, auch wenn sie nicht mehr genutzt werden
- Lösung: Aging

5.3.2.9 NFU mit Aging

- wie NFU, aber andere zyklische Zählerbehandlung
 - Zähler wird zunächst um eine Stelle nach rechts geschiftet (=Division durch 2)
 - danach höchstwertiges Bit auf den Wert des Ref-Bits der Seite gesetzt (Ref-Bit danach gelöscht)
- Beobachtung: bei n-Bit-Zählern ist Zählerwert 0, wenn auf Seite n Zyklen ohne Zugriff
- Zählerwert „altert“, so dass die Wahrscheinlichkeit des Auslagerung steigt, je länger Seite ungenutzt bleibt

5.3.2.10 Zeitpunkt des Einlagerns

bisher:

- „**demand paging**“: Seiten dann eingelagert, wenn sie von Prozess benötigt werden
- Probleme:
 - sehr viele Seitenfehler nach Start oder Phase längerer Verdrängung
 - sehr viele Seitenfehler, wenn Programm mehr Seitenrahmen benötigt als verfügbar sind (z.B. wenn zu viele Programme aktiv sind)

alternativ:

- „**prepaging**“: Seiten werden im Voraus eingelagert, um Prozessen Ausführung mit wenigen Seitenfehlern zu ermöglichen
- Effekt: höherer Durchsatz

5.3.2.11 Thrashing

Thrashing wird als Systemzustand bezeichnet, in dem Anzahl der Seitenfehler stark ansteigt. Dadurch sinkt Durchsatz erheblich, da viel Rechenzeit für Ein- und Ausgabe von Seiten verwendet wird.

Mögliche Ursache: Zu viele Programme gleichzeitig aktiv, so dass für jedes einzelne zu wenig Seitenrahmen bereit stehen.

Möglichkeit zur Vermeidung von Thrashing: spezielle Ein- und Auslagerungsstrategien, die berücksichtigen, dass Prozess zur sinnvollen Ausführung eine bestimmte Menge eingelagerter Seiten benötigt.

5.3.2.12 Working-Set-Verfahren

- WS-Verfahren kombiniert Seitenersetzungsstrategie mit Steuerung der Menge der im Hauptspeicher befindlichen Prozesse
- Konflikt bei:
 - o Seitenersetzungsstrategie will Häufigkeit der Seitenfenster reduzieren
 - o Prozessorverwaltung will Prozessauslastung erhöhen, wozu i.A. Multiprogramminggrad erhöht wird, was Seitenfenster-Häufigkeit erhöht
- Working-Set (Arbeitsmenge): Menge der innerhalb des vergangenen Zeitabschnitts vom Prozess referenzierte Seiten
- Beobachtung: WS ändert sich mit der Zeit, kann dabei Größe ändern
- Generelle Steuerung des WS-Größe durch Wahl der Länge des Zeitabschnitts
- WS-Grundidee:
 - o Nur Seiten, die nicht Teil eines WS sind, dürfen ausgelagert werden
 - o Prozesse werden nur aktiviert, wenn ihr gesamtes aktuelles WS in den Hauptspeicher passt (-> Prepaging)

5.3.3 Weitere Aspekte

5.3.3.1 Gemeinsam genutzte Speicherseiten

- mehrere Prozesse können Speicherbereiche gemeinsam nutzen. Referieren dazu die gleichen Seitenrahmen aus ihren Seitentabellen
- sinnvoll, wenn gleiches Programm mehrfach ausgeführt wird (Code nur einmal im Speicher)
- für Datenaustausch geeignet (Synchronisation beachten!)
- aufwändigere Verwaltung, da Seitenrahmen von mehreren Prozessen genutzt werden

5.3.3.2 Vorrat an freien Seitenrahmen

- schnelles Einlagern nur möglich, wenn freie Seiten vorhanden sind
- möglicher Ansatz:
 - Hintergrundprozess der Seitenverwaltung (paging daemon) untersucht regelmäßig Anzahl der freien Seitenrahmen
 - sinkt diese unter die Grenze, werden Seiten ausgelagert, um einen Vorrat an freien Seitenrahmen zu erhalten

5.4 Segmentierung

5.4.1 Vorüberlegung

- bisher: ein linearer Adressraum pro Prozess
- Problem:
 - Programme haben häufig mehrere, getrennte Bereiche, z.B. Code, Daten, Stack
 - Bereiche können dynamisch wachsen
 - Bereiche haben verschiedene Schutzanforderungen (lesen, schreiben, ausführen)
 - wie im logischen Adressraum unterbringen?
- Lösung: verwenden getrennter Adressräume, sog. **Segmente** für die einzelnen Bereiche

5.4.2 Adressierung und Segmente

- Segment besitzt logischen Adressraum (von 0 bis Obergrenze); Beschreibung:
 - physikalische Startadresse (Basisadresse)
 - ggf. Länge des Segments
 - ggf. Zugriffsrechte für Segment
- Adressierung in Programm erfolgt relativ zu Segmenten; dadurch leichte Verschiebbarkeit der Segmente im physikalischen Speicher
- logische Adresse ist Tupel aus Segment und Position
- MMU setzt logische Adresse in physikalische Adresse um; dabei werden (falls von System unterstützt) Einhaltung der Zugriffsrechte und Segmentgrenze geprüft

5.4.3 Realisierung ohne virtuelle Adressräume

- Segment bildet im physikalischen Speicher zusammenhängenden Abschnitt
- direkte Umsetzung logischer Adressen in physikalische: Addition der Position innerhalb des Segments zur Basisadresse des Segments
- Problem:
 - externe Fragmentierung des physikalischen Speichers
 - vergrößern der Segmente schwierig
 - Segmentgröße durch Größe des physik. Speichers beschränkt
- theoretisch ist Kompaktierung möglich, da Segmente verschiebbar sind, dies ist aber rechenaufwendig

5.4.4 Realisierung mit virtuellen Adressräumen

- Kombiniert Vorteile von Segmentierung und virtueller Adressierung
- zwei Möglichkeiten:
 - ein virtueller Adressraum pro Segment:
 - Segment-Anteil der log. Adresse bestimmt die verwendete Seitentabelle
 - Positions-Anteil ist virt. Adresse im zugeordneten Adressraum
 - alle Segmente in einem virtuellen Adressraum:
 - zunächst Umsetzen der log. Adresse (Segment, Position) in virtuelle (= Segmentbasis+Position)
- danach Umsetzung in physik. Adresse über Seitentabelle

6 Ein- /Ausgabe

Abwickeln von Ein- und Ausgabe ist wichtige Aufgabe des BS. Dabei werden vielfältige Ziele verfolgt, z.B.:

- Verstecken der Details der Programmierung der Hardware
- Vereinheitlichter Zugriff auf Geräte mit gleicher Aufgabe, unabhängig von konkretem Gerät
- Schaffen eines vereinheitlichten Programmiermodells, das möglichst viele Geräteklassen transparent umfasst

6.1 E/A-Hardware

6.1.1 Geräteklassen

zwei grundsätzliche Klassen von Geräten:

- blockierte Geräte (block-devices):
 - o wahlfreier Zugriff auf Datenblöcke fester Größe
 - o Bsp.: Festplatte, CD-Rom
- Zeichenorientierte Geräte (character-devices):
 - o sequentieller Zugriff auf eine Folge von Zeichen
 - o Bsp.: Netzwerk, Drucker, Tastatur, Terminal
- Klassen nicht immer klar:
 - o z.B. Grafikkarte, Uhren

6.1.2 Einbindung

Möglichkeiten zur Einbindung von Geräten ins System:

- häufig über spezielle Steuerhardware (Controller)
 - o spezielle Prozessoren zur Steuerung der Geräte
 - o je nach Controller verschieden stark ausgeprägte „Eigenintelligenz“
- Einbindung an Busse
 - o Am gleichen Bus wie CPU
 - o Sekundäre Busse, die über Brückenbausteine angebunden werden

6.1.3 Kommunikation CPU ↔ Gerät

- CPU sendet Steuerkommandos und Daten an Gerät
- Geräte senden Daten und Statusinformationen an CPU
- Möglichkeiten:
 - o Spezielle Register im Controller, z.B. Datenregister (Ein- und Ausgabe, Steuerregister, Statusregister, ...)
 - o Memory-Mapped: Controller blenden Register in normalen Speicherbereich ein
 - o Interrupts: Geräte senden Unterbrechungssignale an CPU, um Meldungen zu übertragen
 - o DMA: Übertragung von Speicherinhalten zum Gerät und zurück unabhängig von CPU

6.1.3.1 Ein-/Ausgabebefehle

- spezielle Befehle der CPU zum Zugriff auf Register der Controller
- eigener Adressraum für Register -> Schutz vor versehentlichem Zugriff
- Befehle häufig privilegiert -> Schutzfunktion; nur BS kann auf Geräte direkt zureifen

6.1.3.2 Memory-Mapped-I/O

- Geräte eigene Speicherbereiche im physik. Adressraum
- Zugriff über normale Speicherzugriffsbefehle
- Schutz?
- Manchmal Mischformen mit E/A-Befehlen, z.B. Grafikkarte

6.1.3.3 Interrupts

- Geräte bzw. Controller arbeiten unabhängig von CPU
- Asynchrone Beachtung der CPU bei Meldungen vom Gerät (Fehler, Aufgabe erledigt, ...)
- Ablauf:
 - o Gerät erzeugt Interrupts-Signal
 - o CPU unterbricht laufendes Programm
 - o Sprung zu Behandlungsroutine für Interrupt (interrupt-handler), häufig über spezielle Sprungtabelle (Interrupt-Vektor)
 - o Häufig Wechsel von user- in kernel-mode
 - o Dort Bearbeiten des Interrupts
 - o Danach Rücksprung zum unterbrochenen Programm

6.1.3.4 Direkt Memory Acces

- (auch: Busmastering)
- direkter Zugriff auf Speicher durch Geräte
- Geräte übertragen Daten vom und zum Speicher unabhängig von CPU
- Ablauf:
 - o Bereitstellen der Daten bzw. eines leeren Speicherbereichs
 - o Übertragen der Adresse der Daten (plus Steuerinformation) an Gerät oder speziellen DMA-Controller
 - o Übertragen der Daten parallel zum normalen Betrieb der CPU
 - o Nach Abschluss der Daten Nachricht an System (z.B. Interrupt oder Setzen eines Flags im Statusregister)

6.2 E/A-Software

6.2.1 Zugriff auf Hardware

drei prinzipielle Programmiermodelle:

- Programmed I/O
- Interrupts
- DMA

6.2.1.1 Programmed I/O

- CPU schreibt Daten in Register der Geräte/Controller
- CPU prüft in Schleife Statusregister, bis Gerät entsprechende Meldung gibt -> „Polling“
- Problem: aktives Warten; CPU blockiert, bis E/A abgeschlossen

6.2.1.2 Interrupts

- CPU beschreibt Daten- und Steuerregister
- Danach Ausführen anderer Programme bis sich Gerät über Interrupt meldet

6.2.1.3 DMA

- CPU initialisiert Datenübertragung
- danach Ausführen anderer Programme, bis DMA abgeschlossen ist

6.2.2 Schichtenmodell

Mehrere Softwareschichten über der Hardware:

E/A Software im Benutzermodus
Geräteunabhängige E/A-Software im BS
Gerätetreiber
Interrupt-Handler
Hardware

Es folgen die Aufgaben der einzelnen Schichten kurz beschrieben.

6.2.2.1 Interrupt-Handler

- Verarbeiten der Interrupts des jeweiligen Geräte
- Weiterleiten der Meldungen an den Treiber

6.2.2.2 Gerätetreiber

- Bereitstellen einer einheitlichen Schnittstelle im BS
- gerätespezifische Steuerungs- und Anpassungsaufgaben
- Problem: Treiber läuft üblicherweise im Kernel-Mode: wie Einbinden?

6.2.2.3 Geräteunabhängige Software

- einheitliche Schnittstelle für Benutzersoftware
- häufig spezieller Aufruf zur direkten Kommunikation mit Treiber (für Spezialfunktionen der einzelnen Geräte)
- einheitliche Schnittstelle für Treiber
- typische Aufgaben:
 - Abstraktion von Gerätespezifischen Details (z.B. Blockgröße)

Erstellt von Oliver Fleck

Fach: Betriebssysteme

Dozent: Herr M. Friedmann

Zusammenfassung Semester III

- Zwischenspeichern von Daten
- Fehlerbehandlung
- Geräte Belegung und Freigeben

6.2.3 Programmierschnittstelle für Benutzerprogramme

- häufig vollständige Abstraktion von Gerät:
 - einheitlicher Zugriff auf alle Geräte; gleiche Schnittstelle für alles
 - Unterscheidung nur zwischen block- und zeichenorientierten Geräten
 - Meist spezieller Systemaufruf zum direkten Zugriff auf besondere Funktionen der Geräte, von denen nicht abstrahiert wird
- Zwei gängige Programmierparadigmen:
 - Blockierende E/A:
 - § Benutzerprogramm wird unterbrochen, bis E/A-Auftrag abgeschlossen ist
 - § Ggf. Sonderfunktionen wie Timeouts
 - § Meist Mittel der Wahl; bequeme Programmierung
 - Nichtblockierende E/A:
 - § Benutzerprogramm läuft nach E/A-Anforderung parallel weiter
 - § Benachrichtigung an Programm, sobald E/A-Abgeschlossen
 - § Häufig unhandliche Programmierung; eigene Verwaltung von Datenpuffern etc. notwendig
 - § Für Spezialfälle sinnvoll einsetzbar
 - § Vermeidet manchmal den Einsatz von Threads