Band

1

FACHHOCHSCHULE HAGENBERG

Skriptenserie zu Betriebssysteme und Netzwerktechnologie

Betriebssysteme

Zweite Auflage

FACHHOCHSCHULE HAGENBERG

Unterlagen zur Lehrveranstaltung

Stefan Probst
© SSW
Jormannsdorf 99 • A-7431 Bad-Tatzmannsdorf
Telefon +43 699 / 81 50 90 66
E-Mail: stefan.probst@fh-hagenberg.at

Einleitung

Das vorliegende Dokument enthält die Begleitunterlagen zu der Lehrveranstaltung „Betriebssysteme“.

Grundlegende Literatur

Als Grundlage zur Erstellung dieses Werkes wurden ausschließlich folgende Literaturquellen herangezogen und gegebenenfalls zitiert:

- Andrew S. Tanenbaum, Albert S. Woodhull (1997) *Operating Systems: Design and Implementation (2nd Edition)*. ISBN 0-13-638677-6, Prentice Hall, 1997.
- Wikipedia.de – Die freie Enyklopädie im Netz (www.wikipedia.de)

Danksagung

Besonderer Dank gilt Herrn Klaus Wolfmaier und Rudolf Ramler für die Mitarbeit und Mitgestaltung dieses Dokuments.

Verwendung des Dokuments

SYMBOL-LEGENDE	
ⓘ	Informationen
!	Wichtige Hinweise
🖉	Beispiele
?	Verständnisfragen

In diesem Skriptum werden Sie immer wieder auf verschiedene Symbole und Schreibweisen stoßen.

Spezielle Ausdrücke und Begriffe werden durch einfache Anführungszeichen (‚‘) gekennzeichnet. Kommandos werden innerhalb doppelter Anführungszeichen („“) erwähnt. Variable Teile innerhalb eines Kommandos sind *kursiv* gedruckt, genauso wie betonte Wörter. Am Seitenrand werden Sie immer wieder Symbole finden, die auf spezielle Informationen hindeuten.

Inhaltsverzeichnis

Kapitel

1

Grundlagen von Betriebssystemen

Dieses Kapitel beschäftigt sich mit grundlegenden Konzepten und Begriffen von modernen Betriebssystemen.

Computer sind in der heutigen Zeit unabdingbar und das Primärwerkzeug vieler Arbeitszweige. Computer selbst sind jedoch relativ nutzlos ohne Software. Die fundamentalste Software stellt das Betriebssystem dar, dessen Aufgabe in der Verwaltung des Computers und der angeschlossenen Hardware liegt. Tatsächlich macht das Betriebssystem den Computer erst nutzbar. Dieses Kapitel erläutert die Grundlagen moderner Betriebssysteme und zeigt dessen Struktur und Aufbau. Zunächst werden einige grundlegende Begriffe erläutert, die für das Verständnis von Betriebssystemen unablässig sind. Danach wird eine Basisstruktur vorgestellt und näher beleuchtet, die in den gängigsten modernen Betriebssystemen vorzufinden ist. Das Kapitel schließt mit einem solchen Vergleich und stellt die hier vorgestellte Basisstruktur der Struktur von Linux und Windows gegenüber.

Lehrinhalte und -ziele

Dieses Kapitel beschäftigt sich mit Grundlagen von Betriebssystemen und zeigt eine Basisstruktur die in modernen Betriebssystemen vorzufinden ist.

Sie sollten nach diesem Kapitel die Aufgaben und wesentlichen Komponenten eines Betriebssystems, sowie dessen Struktur kennen und erklären können.

Definition von Betriebssystemen

Definition: Betriebssystem

- Betriebssystem (engl.: operating system)

  Die Software, die die Belegung und die Verwendung von Hardwareressourcen (z.B. Arbeitsspeicher, Prozessorzeit, Datenträgerplatz und Peripheriegeräten) steuert. Das Betriebssystem stellt das Fundament dar, auf dem die Anwendungen aufgebaut sind. Zu den bekanntesten Betriebssystemen gehören Mac OS, OS/2, UNIX, Linux, Windows.

  Computer Lexikon Fachwörterbuch, 7. Auflage

Um die Funktionalität von Betriebssystemen zu verstehen, soll zunächst eine Definition der Aufgaben eines solchen erläutert werden. Das Computer Lexikon Fachwörterbuch definiert die Aufgabe von Betriebssystemen wie oben in der Grafik abgedruckt.

Aus der Definition ist ersichtlich, dass das Betriebssystem sich einerseits um das Hardwaremanagement kümmert und andererseits dafür Sorge trägt, dass Anwendungssoftware eine entsprechende Laufzeitumgebung zur Verfügung gestellt bekommt. Dabei wird unterschiedliche Hardware und Herstellerdetails vom Betriebssystem abstrahiert (virtualisiert) damit Anwendungssoftware sich nicht um die Details in Hardwarerealisierungen kümmern muss.

Aufgaben des Betriebssystems

Aufgaben des Betriebssystems

- Hardwaremanagement
  - Initialisierung und Test der Hardware
  - Hardware betreiben und überwachen
  - Kontrolle der Eingabe- und Ausgabeoperationen
- Softwaremanagement
  - Prozessverwaltung
  - Speicherverwaltung
  - Laden und Ausführen benötigter Programme
- Netzwerkmanagement
  - Abstraktion der Verteilung von Hard- und Software
- Datenmanagement
  - Abstraktion und Strukturierung des Datenspeichers
- Benutzerschnittstelle

Wie bereits aus der vorigen Definition gesehen, hat das Betriebssystem eine Reihe von Aufgaben zu bewältigen, die vorwiegend mit Hardware und Anwendungssoftware zu tun haben. Im Detail können die Aufgaben auf die folgenden Bereiche aufgegliedert werden.

Hardwaremanagement

Diese Komponente sorgt dafür, dass die einzelnen am System angeschlossenen Hardwarekomponenten im System und den einzelnen Anwendungen zur Verfügung gestellt werden. Dabei abstrahiert das Betriebssystem die Herstellerdetails und stellt den Anwendungen eine einheitliche Schnittstelle zum Zugriff auf die Hardware bereit. Das Betriebssystem ist weiters dafür verantwortlich, der Hardware die benötigten Ressourcen bereitzustellen und überwacht und koordiniert die Operationen der Hardware. Gerade letzteres ist wichtig, da oftmals verschiedene Applikationen zur gleichen Zeit auf die gleiche Hardware zurückgreifen (z.B. Festplatte, Tastatur) und es insofern jemanden geben muss, der eventuell entstehende Konflikte vermeidet.

Softwaremanagement

Eine der wesentlichsten Aufgaben eines Betriebssystems ist das Softwaremanagement. Diese Komponente trägt dazu bei, dass einzelne Anwendungsprogramme in einem geordneten Umfeld arbeiten können. Dazu stellt das Betriebssystem den Anwendungen die benötigten Ressourcen wie z.B. Speicher bereit. Um den Anwendungsentwickler nicht unnötig mit den Details der Hardware zu konfrontieren, stellt das Betriebssystem jeder Anwendung eine virtuelle Umgebung bereit. Damit sorgt das Betriebssystem, das sich einzelnen Anwendungen nicht gegenseitig behindern.

Netzwerkmanagement

Ein Aspekt der gerade bei modernen Betriebssystemen immens an Bedeutung gewinnt, ist die Fähigkeit des Betriebssystems die Verteilung über ein Netzwerk zu berücksichtigen beziehungsweise zu abstrahieren. War es vor wenigen Jahren noch ausreichend, Netzwerkunterstützung und die Verteilung von Daten in einem Netzwerk im Betriebssystem zu berücksichtigen, so befassen sich aktuelle Arbeiten mit der Möglichkeit, ein Betriebssystem über die Rechnergrenze hinweg zu verteilen. Hierbei realisiert eine Softwareschicht (die so genannte Middleware) einen gemeinsam genutzten Speicher und bietet allen Kontenpunkten Dienste an mit der Zielsetzung eine vollständige Verteilungstransparenz zu erreichen.

Datenmanagement

Die Komponenten des Datenmanagements kümmert sich um die Persistierung von Daten auf unterschiedliche Datenspeicher. Bei immer größer werdenden Datenspeicher und unterschiedlichen Speichermedien (z.B. Diskette, Festplatte, CD, DVD, USB-Stick) ist dies kein triviales Problem, vor allem wenn Zuverlässigkeit und effiziente Zugriffszeiten gewährleistet werden sollen. Deshalb wird man heute in der Praxis oftmals mit unterschiedlichsten Dateisystemen konfrontiert, die in der Regel nur sehr schlecht miteinander zusammenarbeiten. Oftmals muss zwischen Zuverlässigkeit und Effizienz des Dateisystems gewählt werden.

Benutzerschnittstelle

Alle zuvor erwähnten Komponenten arbeiten im Hintergrund und ermöglichen das Arbeiten des Benutzers mit dem Computer. Im optimalen Fall nimmt der Benutzer die Tätigkeiten dieser Komponenten gar nicht wahr. Allerdings stellt das Betriebssystem eine Schnittstelle für den Benutzer bereit, über die er mit dem Betriebssystem und den Anwendungen kommunizieren kann. Eine solche Schnittstelle reicht von der einfachen Positionierung von Zeichen auf einem Textbildschirm bis hin zu einem vollgraphischen, hochauflösenden Fenstersystem.

Begriffe und Definitionen

Grundbegriffe

- Prozess
  - Abstraktion eines laufenden Programms
    - Besteht aus einem Programm, Eingaben, Ausgaben und Zuständen
  - Besteht aus zwei Teilen
    - Programm, das im Hauptspeicher geladen ist
    - Prozessumgebung, in der korrektes Ablaufen ermöglicht wird (vom Betriebssystem zur Verfügung gestellt)
  - Eigenschaften
    - Programm kann in mehrere Prozesse aufgeteilt sein
    - Prozess besitzt eigene Umgebung und ist abgeschottet von anderen Prozessen

Zunächst sollen nun wichtige Begriffe erläutert werden, die mit Betriebssystemen eng zusammenhängen.

Prozess

Ein auf dem System befindliches Programm das gerade ausgeführt wird nennt man Prozess. Innerhalb eines Ablaufs werden dem Prozess Eingaben zugestellt, Ausgaben produziert und Daten verarbeitet. Dabei durchwandert ein Prozess in der Regel eine Reihe von Zuständen (die so genannten Prozesszustände).

Prinzipiell besteht ein Prozess aus zwei Teilen. Der erste Teil ist das Programm, das in den Hauptspeicher geladen ist und die einzelnen auszuführenden Instruktionen enthält. Damit ein solches Programm korrekt laufen kann, wird vom Betriebssystem eine Prozessumgebung bereitgestellt, die jeweils die benötigten Ressourcen zur Verfügung stellt.

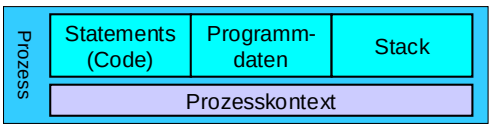
Eigenschaften

Ein bedeutendes Detail bei Prozessen ist das Faktum, dass ein Programm in mehrere Prozesse aufgeteilt sein kann. Gerade in Bezug auf den Aspekt in verteilten Softwaresystemen ist diese Eigenschaft mit zusätzlichen Anforderungen an das Betriebssystem verbunden, da Prozesse nun gemeinsame Speicherbereiche teilen müssen bzw. miteinander kommunizieren müssen (Interprozess-Kommunikation). Auch dafür muss das Betriebssystem entsprechende Mechanismen bereitstellen.

Ein weiterer wichtiger Punkt ist, dass Prozesse über eine eigene virtuelle Umgebung verfügen und somit von anderen Prozessen abgeschottet sind. Diese Abschottung ist speziell aus Gründen der Sicherheit unabdingbar.

Grundbegriffe

- Adressraum eines Prozesses
  - Code (Anweisungen, Statements)
  - Programmdaten
    - Speicher für Variablen, dynamisch angeforderten Speicher (Heap), Konstanten
  - Stack
    - Speicher für Parameterübergabe an Routinen
    - Rücksprungadresse nach Routine
- Prozesskontext
  - Statusinformationen
    - CPU-Register
    - Zustand des Stacks und Programmdaten
    - Laufzeitinformation



Adressraum

Jedem Prozess wird vom Betriebssystem ein Adressraum zugeordnet. Dieser Adressraum ist eine Liste von Speicherzellen die ein Prozess lesen und schreiben kann. Der Adressraum enthält folgende Daten des Prozesses:

- Code: Das ausführbare Programm bzw. dessen Instruktionen
- Programmdaten: Die im Programm verwendeten Variablen, Konstanten und der dynamisch angeforderte Speicher (Heap).
- Stack: Speicher für Parameterübergabe an Routinen und Rücksprung-adressen.

Prozesskontext

Zusätzlich zum Adressraum führt das Betriebsystem für jeden Prozess auch Statusinformationen mit. Diese werden im so genannten Prozesskontext gehalten. Hier befindet sich eine Reihe von dem Prozess zugeordneten CPU-Registern, der Zustand des Stacks und der Programmdaten (z.B. Stack Pointer) und Laufzeitinformation (z.B. Program Pointer).

Grundbegriffe

- Multiuser-Betriebssystem
  - Mehrere Benutzer können abgeschottet voneinander parallel mit dem Betriebssystem arbeiten
- Multiprozess-Betriebssystem (Multitasking)
  - Mehrere Prozesse werden auf einer CPU ausgeführt
  - Prozesse „teilen“ sich CPU
  - Pseudosimultane Ausführung von Prozessen
- Multiprozessor-Betriebssystem
  - Hardware verfügt über mehrere Prozessoren
  - Prozesse werden auf verfügbare Prozessoren aufgeteilt
  - Simultane Ausführung von Prozessen

Betriebssysteme werden je nach ihrer Fähigkeit mehrere Prozesse und Benutzer gleichzeitig zu unterstützen in verschiedene Kategorien eingeteilt.

Multiuser-Betriebssysteme

In dieser Kategorie werden Betriebssysteme eingeordnet die das parallele Arbeiten mehrerer Benutzer auf einer Maschine unterstützen. Dabei ist es unerheblich ob die Benutzer physisch auf derselben Maschine arbeiten oder über das Netzwerk Prozesse auf der Maschine starten. Wichtig ist jedoch, dass die Benutzer voneinander abgeschottet arbeiten können. Insofern muss das Betriebssystem jeden Benutzer eine eigene virtuelle Umgebung bereitstellen.

Multiprozess-Betriebssysteme

In dieser Kategorie werden Betriebssysteme eingeordnet, die eine parallele Ausführung mehrerer Prozesse auf *einer* CPU erlauben. Um dies gewährleisten zu können, muss das Betriebssystem über Mechanismen verfügen, die CPU auf die verschiedenen Prozesse so aufzuteilen, dass der Benutzer den Eindruck erhält diese würden parallel laufen (Pseudosimultan). Tatsächlich ist jedoch zu einem bestimmen Zeitpunkt immer nur ein Prozess aktiv.

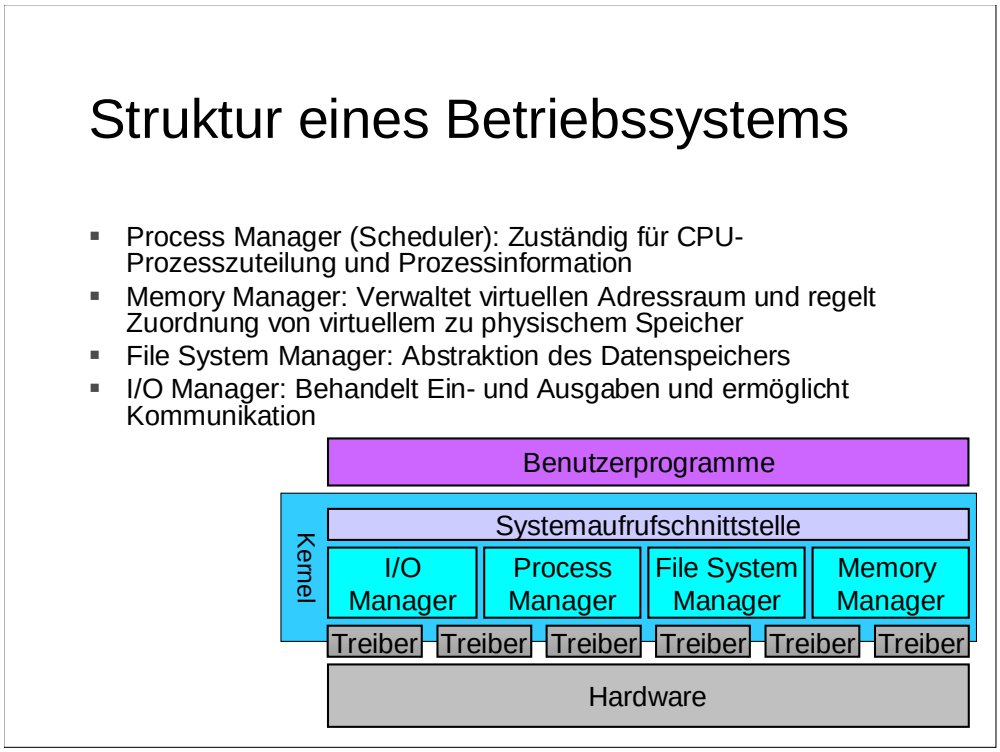
In der Praxis gibt es unterschiedliche Techniken wie diese Pseudosimultanität realisiert wird. Dies kann einerseits durch freiwilliges Abgeben der CPU durch den Prozess geschehen (nonpreemtive Scheduling) oder durch die Kontrolle des Betriebssystems (preemtive Scheduling). In der Praxis hat sich der letztere Ansatz bewährt.

Multiprozessor-Betriebssysteme

In dieser Kategorie werden Betriebssysteme eingeordnet, die Systeme mit mehreren Prozessoren unterstützen. Solche Betriebssysteme sind wirklich in der Lage, mehrere Prozesse gleichzeitig auszuführen, da die vorhandenen Prozessoren gleichzeitig verschiedene Instruktionen (von jeweils unterschiedlichen Prozessen) ausführen können. Die Anforderung an ein solches Betriebssystem ist die möglichst effiziente Ausnutzung der vorhandenen Rechenleistung. Bei der Aufteilung der Prozesse auf die vorhandenen CPUs müssen Konflikte (z.B. gleichzeitiger Zugriff von unterschiedlichen Prozessen auf die gleiche Hardware) und gegenseitige Ausschlüsse (z.B. Prozess 1 kann nur weiterarbeiten wenn Prozess 2 Daten liefert, dieser wartet jedoch auf die Freigabe einer Hardwareressource die Prozess 1 noch belegt) verhindert werden.

Die hier aufgeführte Kategorisierung ist nicht exklusiv, d.h. ein Betriebssystem kann in mehrere Kategorisierungen fallen. Tatsächlich findet man heute fast ausschließlich nur mehr Betriebssysteme, die allen drei Kategorisierungen gerecht werden (Multiuser/Multitasking/Multiprozessor-OS).

Struktur eines Betriebssystems



Analysiert man die Struktur und den Aufbau eines Betriebssystems, so stellt man fest, dass diese alle über eine ähnliche Struktur verfügen. In weiterer Folge sollen nun die üblichen Komponenten eines Betriebssystems kurz erläutert werden. Detaillierte Informationen folgen dann in den weiteren Folgekapiteln.

Process Manager

Der Process Manager (oft auch *Scheduler*) genannt, ist zuständig für die Zuteilung von CPU-Zeit an die einzelnen Prozesse. Zudem stellt und aktualisiert dieser die Prozessinformationen indem er ständig eine Liste mit aktiven Prozessen im System mitführt. Insofern kann gesagt werden, dass der Process Manager die Koordinationsrolle im System in Bezug auf die Ausführung von Prozessen übernimmt.

Memory Manager

Der Memory Manager arbeitet eng mit dem Process Manager zusammen und ist für die Bereitstellung von Speicher für die einzelnen Prozesse verantwortlich. Dabei bekommt jeder Prozess einen eigenen, virtuellen und linearen Speicherbereich. Der Memory Manager ordnet diese virtuellen Speicheradressen dem physikalischen Speicherbereich zu. Da der virtuelle Speicherbereich oftmals viel größer ist als der tatsächlich vorhandene physikalische Speicher, verfügt der Memory Manager über die Möglichkeit, Speicherbereiche auf externe Medien (z.B. Festplatte) auszulagern.

File System Manager

Die Aufgabe des File System Managers ist die Abstraktion der vom Betriebssystem verwalteten Festspeicher. Zum einen muss dabei die Dauer des Speichers (Wechselmedium, eingebauter Speicher), zum anderen die Zugriffsart (Lesen und Schreiben, nur Lesen, nur 1x Schreiben) und das Zugriffsmedium berücksichtigt

werden. Zudem haben unterschiedliche Programme und Systeme unterschiedliche Anforderungen. Insofern unterstützen die gängigen Betriebssysteme heute mehrere verschiedene Dateisysteme, die sich in deren Funktionalität voneinander unterscheiden (z.B. FAT, NTFS, ext2, ext3, reiserfs).

I/O Manager

Der I/O Manager verwaltet die Hardware und ermöglicht Prozessen mit angeschlossenen Geräten zu kommunizieren. Dabei werden Ein-/ und Ausgaben an Prozesse zugestellt und die Datenstrom gesteuert. Zudem ist der I/O Manager dafür verantwortlich, Prozesse die auf Eingaben warten in einen Wartezustand zu versetzen und diese wieder aufzuwecken sobald die Daten verfügbar sind. Darüber hinaus trägt der I/O Manager dafür Sorge, dass es zu keinen Konflikten beim Zugriff auf die Hardware durch unterschiedliche Prozesse kommt.

In weiterer Folge sollen nun die Aufgaben der einzelnen Komponenten erläutert werden.

Process Manager

Process Manager

- Koordiniert alle Aufgaben die bei der Ausführung eines Prozesses anfallen
  - Management des Lebenszyklus: Erstellung, Ausführung und Beendigung des Prozesses
  - Scheduling: Erstellung eines Ausführungsplans für (pseudo-)simultane Prozesse
  - Context-Switching: Übergabe der CPU an anderen Prozess
  - Timing: Überwachung der Ausführungszeit eines Prozesses
  - Speicherverwendung: Koordination mit Memory Manager (Anfordern, Freigeben von Hauptspeicher)
  - Koordination mit Dateisystem: Ein-/Ausgabe von Prozessen, Ladeoperation des Prozess
  - Interprozesskommunikation: Zustellen von Nachrichten zwischen Prozessen und Betriebssystem

Der Process Manager koordiniert die Aufgaben, die bei der Ausführung eines Prozesses anfallen. Dazu zählen:

- Management des Lebenszyklus: Jeder Prozess unterliegt einem Lebenszyklus, der von der Erstellung über die Ausführung bis hin zur Beendigung reicht. In jeder Prozessphase sind verschiedene Dinge vom Betriebssystem zu erledigen, die vom Process Manager durchgeführt werden.

- Scheduling: Um Prozesse parallel ausführen zu können, muss das System einen Ausführungsplan der aktiven Prozesse warten und durchsetzen.

- Context-Switching: Wird die CPU an einen anderen Prozess übergeben, so sind gewisse Aufgaben notwendig (z.B. Bereitstellen der benötigten Ressourcen für den neuen Prozess, Bereitstellung des virtuellen Speicherbereichs, Aktualisierung des Prozesskontexts).

- Timing: Überwachung der korrekten Ausführung des Prozesses innerhalb seines Ausführungsplans.

- Speicherverwendung: Steuerung des Memory Manager so dass der gerade aktive Prozess seinen virtuellen Speicherbereich vorfindet.

- Koordination mit Dateisystem: Steuerung des I/O Manager, so dass vom Prozess benötigte Dateioperationen durchgeführt werden können.

- Interprozesskommunikation: Steuerung und Bereitstellung von Kommunikationsmitteln zu anderen Prozessen.

Memory Manager

Memory Manager

- Verwaltung der Speicherhierarchie
  - Cache
  - RAM
  - Festspeicher

Größe Kosten, Geschwindigkeit

- Aufgaben
  - Verwaltung des Speichers (Liste mit freien, benutzten Speicherbereichen)
  - Stellt Prozessen Speicherbereiche zur Verfügung und gibt diese nach Beendigung wieder frei
  - Erweiterung des Hauptspeichers durch Auslagerung von Speicherbereichen auf die Festplatte (swapping)

Der Memory Manager ist für die Verwaltung des Speichers zuständig. Dabei sorgt er dafür, dass der virtuelle Speicherbereich der jedem Prozess zugesichert ist, real zur Verfügung steht. Da dieser Speicherbereich meist größer ist als der physisch vorhandene Speicher stellt der Memory Manager verschiedene Mechanismen bereit um dieses Defizit auszugleichen. Dabei greift der Memory Manager auf die im System vorhandene Speicherhierarchie zu. Die Kernproblematik dabei liegt in der effizienten Nutzung dieser Hierarchie, da sich die zur Verfügung stehenden Speicher stark voneinander in Hinblick auf Größe, Kosten und Geschwindigkeit unterscheiden. So ist der schnellste zur Verfügung stehende Speicher (Cache) meist sehr klein und sehr teuer, der größte zur Verfügung stehende Speicher (Festplatte) meist sehr groß und relativ günstig, dafür sehr langsam.

Zu den Aufgaben des Memory Managers gehört die Verwaltung des physikalischen Speichers. Dazu führt er eine Liste mit freien und belegten Speicherbereichen. Fordern Prozesse Speicherbereiche an, so stellt der Memory Manager diese zur Verfügung und gibt diese nach der Beendigung des Prozesses wieder frei. Da Prozesse auf Grund der ihnen zur Verfügung stehenden virtuellen Speicherbereiche meist mehr Speicher anfordern als real zur Verfügung steht, kann der Memory Manager auf Hintergrundspeicher wie z.B. die Festplatte zurückgreifen und somit den realen Hauptspeicher erweitern.

File System Manager

File System Manager

- Verwaltung von Festspeichern
  - Laden / Speichern von Programmen, Daten und Information
  - Große Informationsmengen müssen verwaltbar sein
- Abstrahiert und strukturiert Festspeicher
  - Abbildung von logischer Struktur (z.B. Dateien und Verzeichnisse) auf physikalische Struktur (Geometrie des Speichermediums)
- Koordination zwischen parallel zugreifende Prozesse
  - Integrität und Konsistenz der Daten muss gewährleistet bleiben
  - Gegenseitiges Überschreiben ausschließen
  - Gegenseitigen Ausschluss (Deadlock) vermeiden

Der File System Manager ist für die Verwaltung der im System zur Verfügung stehenden Festspeicher zuständig. Zum einen muss dabei die Dauer des Speichers (Wechselmedium, eingebauter Speicher), zum anderen die Zugriffsart (Lesen und Schreiben, nur Lesen, nur 1x Schreiben) und das Zugriffsmedium berücksichtigt werden.

Diese Festspeicher werden zum Laden und Speichern von Programmen, deren Daten und Informationen verwendet. Dabei ist es wichtig, dass sowohl kleine als auch sehr große Informationsmengen ablegbar und verwaltbar sind. Gerade der damit verbundene Verwaltungsaufwand hat in den letzten Jahren unterschiedliche Ansätze und Dateisysteme hervorgebracht.

Um den Festspeicher effektiv nutzen zu können muss dieser eine gewisse Struktur aufweisen. Dabei werden die physikalischen Eigenschaften des Mediums (Geometrie des Speichermediums) auf eine logische Struktur abgebildet (z.B. Dateien und Verzeichnisse). Diese Struktur wird auch Dateisystem genannt.

Ein weiterer wichtiger Aspekt ist die Koordination der Zugriffe von mehreren parallel laufenden Prozessen. Dabei muss der File System Manager die Integrität und Konsistenz der Daten gewährleisten. Dies ist insbesondere dann schwierig, wenn sehr viele Prozesse gleichzeitig auf die gleiche Datei zugreifen und diese ggf. ändern (z.B. Datenbank-Dateien). Dabei muss ein gegenseitiges Überschreiben genauso verhindert werden wie ein möglicher gegenseitiger Ausschluss (Deadlock). Bei einem gegenseitigen Ausschluss wird davon gesprochen, wenn zwei Prozesse in einen gegenseitigen Konflikt geraten, der von den Prozessen selbst nicht mehr aufgelöst werden kann.

I/O Manager

I/O Manager

- Verwaltung von Ein-/Ausgabegeräten
  - Hardwaresteuerung der Geräte
  - Interrupt und Fehlerbehandlung
- Kooperation von Geräten
  - Koordination von Datenflüssen zwischen Geräten und Betriebssystem
  - Effiziente Steuerung von Datenflüssen je nach Gerättyp
    - Blockorientierte Geräte
    - Zeichenorientierte Geräte
- Unterstützt Betriebssystem bei Tätigkeit
  - Dienste für Prozess Manager, Memory Manager und File System Manager

Der I/O Manager ist für die Steuerung und Verwaltung der Ein-/Ausgabegeräte zuständig und stellt diese den einzelnen Prozessen zur Verfügung.

Der I/O Manager verwaltet einerseits die Hardware in dem er direkt für die Gerätesteuerung, die Interrupt und Fehlerbehandlung zuständig ist, zum anderen koordiniert dieser auch die Datenflüsse zwischen den Geräten und dem Betriebssystem. Dabei unterscheidet der I/O Manager zwei Gerättypen, die *blockorientierten* und die *zeichenorientierten* Geräte. Von einem blockorientiertem Gerät können jeweils nur immer ganze Datenblöcke gelesen oder geschrieben werden, ein zeichenorientiertes Gerät liefert einen seriellen Datenstrom.

Da eine der Hauptaufgaben eines Betriebssystems die Verwaltung von Hardware ist und auch die anderen Komponenten des Betriebssystem auf die Hardware zurückgreifen müssen (z.B. Process Manager auf die CPU, Memory Manager auf den Speicher, File System Manager auf die Festplatte), stellt der I/O Manager eine zentrale und kritische Komponente in jedem Betriebssystem dar. Allerdings ist die Realisierung eines I/O Managers recht komplex, da unterschiedliche Hardware (die meist zum Zeitpunkt der Erstellung des Betriebssystems gar nicht bekannt ist) in das Betriebssystem eingegliedert werden muss. Zudem sollten Fehlerzustände (evt. durch unsaubere Treiber hervorgerufen) behandelt werden und eventuell übergangen werden (*Fail-Over*).

Systemaufrufschnittstelle

Systemaufrufschnittstelle

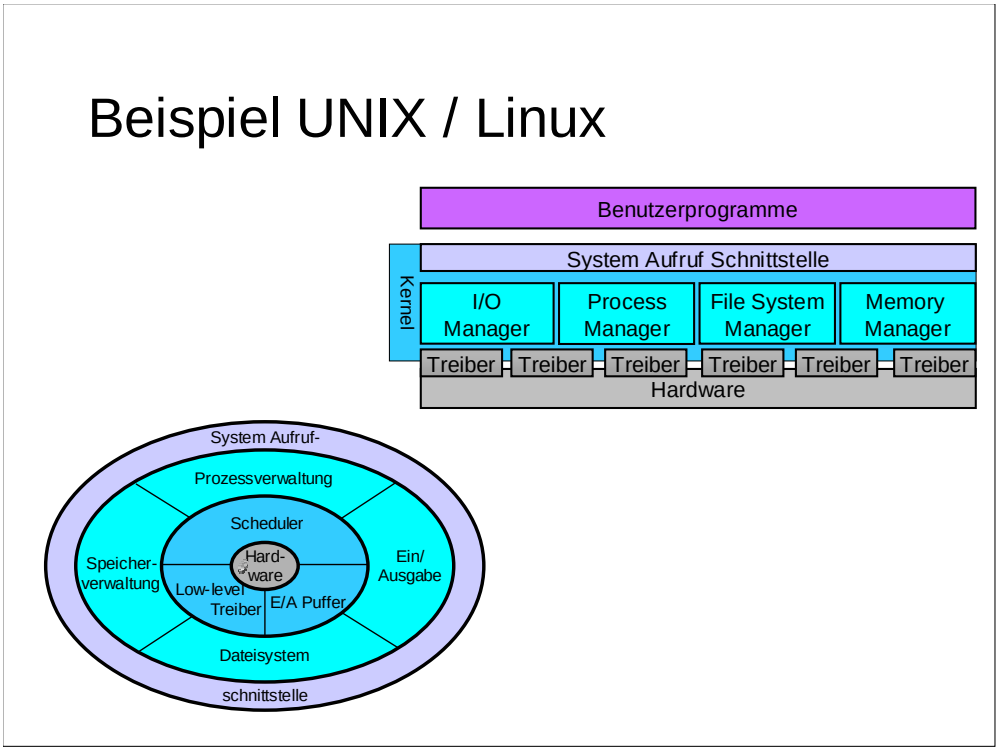
- Ermöglicht Prozessen auf Betriebssystemdienste zurückzugreifen
  - Benutzerprogramme laufen in einer virtuellen Maschine
    - Betriebssystem stellt Prozess eine virtuelle Maschine zur Verfügung
  - Benutzerprogramme können nicht direkt auf Hardware zugreifen sondern interagieren mit virtueller Maschine
  - Systemaufrufschnittstelle ermöglicht Benutzerprogrammen auf Dienste des Betriebssystems nutzen zu können
    - Stellt Programmroutinen für Betriebssystemfunktionen bereit
    - Programmierschnittstelle zum Betriebssystem (API)

Prozesse laufen normalerweise im so genannten Benutzermodus (*user mode*). Die Privilegien im Benutzermodus sind sehr eingeschränkt. Benutzerprogramme laufen in diesem Modus in einer virtuellen Maschine, die vom Betriebssystem zur Verfügung gestellt wird. Diese virtuelle Maschine abstrahiert die Eigenheiten der Hardware vollkommen vor dem Prozess und lässt den Prozess glauben, dass dieser exklusiv läuft. Tatsächlich kann der Prozess allerdings nicht direkt auf die Hardware zugreifen sondern interagiert lediglich mit einer Virtualisierung der Hardware. Dies hat den Vorteil, dass kein Programm z.B. durch einen Fehler das System zum Absturz bringen kann.

Um trotzdem die Dienste des Betriebssystems nutzen zu können (z.B. Zugriff auf die Festplatte zum Speichern einer Datei) muss das Benutzerprogramm Kernelfunktionen aufrufen. Dazu stellt das Betriebssystem eine *Systemaufrufschnittstelle* bereit, über die Dienste des Betriebssystems Benutzerprogrammen zugänglich gemacht werden.

Die Systemaufrufschnittstelle stellt dabei eine Reihe von Funktionen und Routinen zur Verfügung, die von Benutzerprogrammen genutzt werden können. Ein solcher Funktionsaufruf (*system call*) bewirkt einen Sprung in den privilegierten Teil des Systems (*kernel mode*), wo dieser dann mit erhöhten Rechten ausgeführt wird. Insofern ist die Systemaufrufschnittstelle die Programmierschnittstelle zum Betriebssystem.

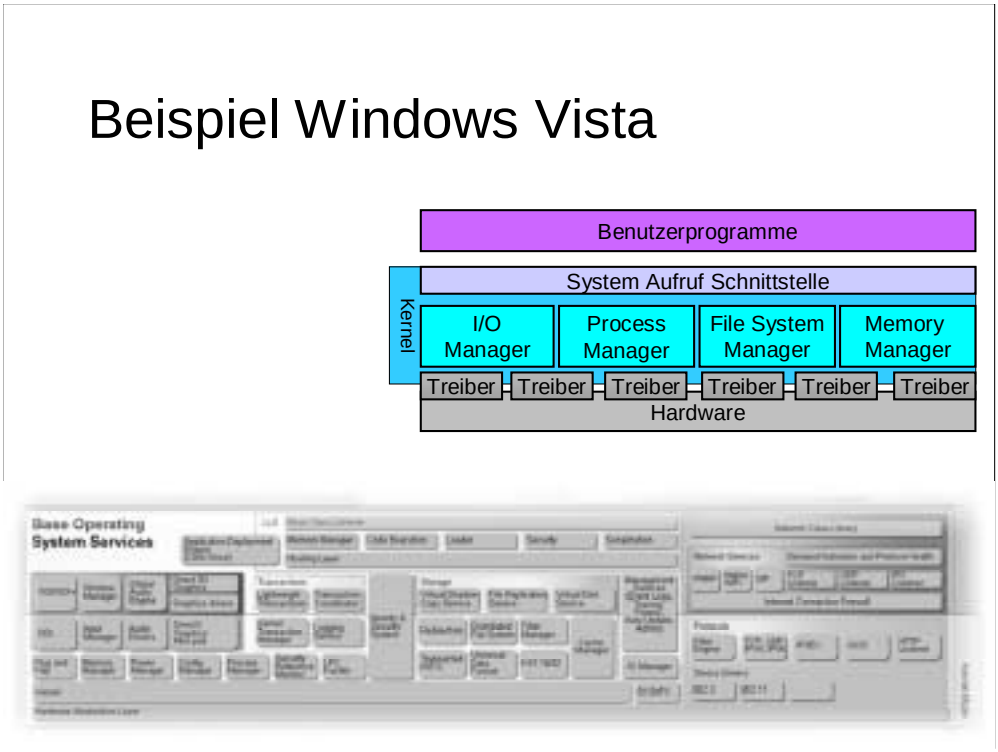
Fallbeispiel UNIX, Linux



Abschließend soll nun anhand von Fallbeispielen die Umsetzung der in diesem Kapitel vorgestellten Basisstruktur von Betriebssystemen veranschaulicht werden.

Im Falle von UNIX wird der Kernel direkt von einer Systemaufrufschnittstelle umgeben, die jeweils die Schnittstelle zwischen Kernel und Benutzerprogrammen darstellt. Der Kernel selbst stellt Module zur Prozessverwaltung (Process Manager), Ein-/Ausgabe (I/O Manager), Dateisystem (File System Manager) und Speicherverwaltung (Memory Manager) bereit. Diese umschließen das Herz des Kernels, der zum einen aus dem Scheduler besteht (Teil des Process Managers), den Low-level Treibern die jeweils die Verbindung zur Hardware herstellen und einem E/A-Puffer, der Ein-/Ausgabedaten zwischenspeichert bis diese von den entsprechenden Prozessen verarbeitet werden.

Fallbeispiel Windows Vista



Auch Windows verfügt über alle vorgestellten Komponenten. Die Systemaufrufschnittstelle nennt sich in Windows Vista WinFX und bietet .NET-Klassen und Komponenten zur Interaktion mit dem Betriebssystem an. WinFX löst die bis dato bekannte und etwas komplexe Win32 API ab. Sowohl I/O Manager, Process Manager, File System Manager und Memory Manager finden sich in den Base Operating System Services wieder. Neu ist die Abkoppelung von Netzwerk- und Kommunikationskomponente, die nun als Windows Communications Foundation (WCF, früher Indigo) bekannt sind. Auch die Visualisierung wurde vom Betriebssystem abgekoppelt und nennt sich nun Windows Presentation Foundation (WPF, früher Avalon). Darüber hinaus soll in Zukunft der File System Manager eine datenbankbasierte Strukturierung mit Metadaten ermöglichen (WinFS).

Kapitel

2

Der Process Manager

Dieses Kapitel erläutert den Process Manager und dessen Aufgaben im Detail.

Die Hauptaufgabe des Betriebssystems ist es, dafür zu sorgen, dass die laufenden Prozesse korrekt verwaltet und dessen Lebenszyklus durch das Betriebssystem unterstützt wird. Da im Normalfall immer mehrere Prozesse scheinbar gleichzeitig laufen sollen, muss das Betriebssystem über Mechanismen und Strategien verfügen, die Ausführung der Prozesse zu planen und verwalten. Die hier erwähnten Aufgaben werden vom Process Manager durchgeführt. Dieses Kapitel erläutert die Aufgaben und die Funktion dieser Komponente. Dabei wird hauptsächlich auf die unterschiedlichen Strategien zur Erreichung von Multitasking und deren Eigenschaften eingegangen.

Lehrinhalte und -ziele

Dieses Kapitel beschäftigt sich mit dem Process Manager, dessen Aufgaben und Funktionalität. Dabei werden die unterschiedlichen Möglichkeiten der Realisierung in Form von Strategien und Algorithmen angesprochen und deren Umsetzung im Betriebssystem gezeigt.

Sie sollten nach diesem Kapitel über die Aufgaben des Process Managers und deren Umsetzung in Betriebssystemen bescheid wissen. Vor allem sollten Sie die unterschiedlichen Strategien des Schedulers kennen und deren Vor- und Nachteile. Darüber hinaus sollten Sie die Zusammenhänge zwischen den Process Manager und den Prozessen kennen und den Lebenszyklus eines Prozesses in einem Betriebssystem in Zusammenhang mit den anderen beteiligten Betriebssystem-komponenten skizzieren können.

Aufgaben des Process Managers

Process Manager

- Koordiniert alle Aufgaben die bei der Ausführung eines Prozesses anfallen
  - Management des Lebenszyklus: Erstellung, Ausführung und Beendigung des Prozesses
  - Scheduling: Erstellung eines Ausführungsplans für (pseudo-)simultane Prozesse
  - Context-Switching: Übergabe der CPU an anderen Prozess
  - Timing: Überwachung der Ausführungszeit eines Prozesses
  - Speicherverwendung: Koordination mit Memory Manager (Anfordern, Freigeben von Hauptspeicher)
  - Koordination mit Dateisystem: Ein-/Ausgabe von Prozessen, Ladeoperation des Prozess
  - Interprozesskommunikation: Zustellen von Nachrichten zwischen Prozessen und Betriebssystem

Das korrekte Ausführen von Prozessen ist eine der Hauptaufgaben eines Betriebssystems. Gerade aber die Möglichkeit moderner Betriebssysteme mehrere Prozesse scheinbar simultan auszuführen macht diese Aufgabe zu einem nicht trivialen Task. Der Process Manager ist dafür verantwortlich, dass mehrere Prozesse reibungslos ablaufen können und stellt diesen eine entsprechende Laufzeitumgebung zur Verfügung. Dabei koordiniert er alle Aufgaben, die bei der Ausführung von Prozessen anfallen.

Management des Lebenszyklus

Jeder Prozess unterliegt einem Lebenszyklus, wobei jeweils eigene Tätigkeiten durch das Betriebssystem durchgeführt werden müssen. Bei der Erzeugung eines Prozesses muss dieser z.B. mit den benötigten Ressourcen versorgt werden und das Betriebssystem erzeugt einen Prozesskontext, der diesem Prozess zugewiesen wird. Dieser Prozesskontext muss laufend während der Ausführung des Prozesses aktualisiert werden. Wird der Prozess beendet, werden die vom Prozess in Anspruch genommenen Ressourcen wieder freigegeben und der Prozesskontext gelöscht.

Scheduling

Moderne Betriebssysteme bieten die Möglichkeit, mehrere Prozesse scheinbar parallel auszuführen. Tatsächlich kann aber jeweils pro CPU nur ein Prozess zu einem bestimmten Zeitpunkt aktiv ausgeführt werden. In der Praxis erhalten die aktiven Prozesse jeweils kurze Zeiteinheiten, in der sie die CPU nutzen können. Damit scheint es so, als würden diese Prozesse parallel ausgeführt werden. Die Zuteilung von CPU-Zeiteinheiten an die aktiven Prozesse bedarf jedoch einer sorgfältigen Planung. Deshalb gibt es unterschiedliche Strategien, wie diese

Zuteilung erfolgen kann. Diese Strategien werden später in diesem Kapitel besprochen.

Context Switching

Wie zuvor erwähnt, teilen sich die Prozesse die im System vorhandene CPU. Wird die CPU an einen anderen Prozess übergeben, so müssen vom Betriebssystem einige Managementaufgaben erledigt werden. Dazu zählt das Bereitstellen der benötigen Ressourcen für den neuen Prozess, die Bereitstellung des virtuellen Speicherbereichs, der eventuell noch vom Festspeicher eingelagert werden muss bzw. die Auslagerung von nicht mehr benötigten Speicherbereichen und die Aktualisierung des Prozesskontexts. Das Ein- und Auslagern von Speicherbereichen wird im Kapitel 3 näher behandelt.

Koordination der Ausführung von Prozessen

Nachdem Prozesse durch den Scheduler in den Ausführungsplan aufgenommen wurden, muss der Process Manager auch die Ausführung der einzelnen Prozesse überwachen und koordinieren. Dazu verfügt der Process Manager über einen *Timing*-Mechanismus, der den korrekten Ablauf nach Ausführungsplan der einzelnen Prozesse überwacht.

Zudem koordiniert der Process Manager auch die Zusammenarbeit mit den anderen Betriebssystemkomponenten wie z.B. dem Memory Manager, der dafür sorgt, dass die jeweils dem aktiven Prozess zugehörigen virtuellen Speicherbereiche verfügbar sind oder dem File System Manager, damit die vom Prozess getätigten Dateioperationen korrekt durchgeführt werden können.

Interprozesskommunikation

Da Programme oftmals aus mehr als einem Prozess bestehen, muss das Betriebssystem eine Möglichkeit zur Kommunikation zwischen diesen Prozessen zur Verfügung stellen. Dazu bietet die Systemaufrufschnittstelle eine Reihe von Mechanismen zur Interprozesskommunikation (IPC), mit deren Hilfe es möglich ist, einfache oder auch komplexe Nachrichten zwischen den Prozessen im Betriebssystem zuzustellen.

Der Scheduler

Scheduler

- Entscheidet, wie die CPU-Zuteilung zu den einzelnen Prozessen erfolgen soll
  - Ablaufplan
- Strategie, nach der Ablaufplan erstellt wird nennt man *Scheduling*
  - Nonpreemptive: Prozess steuert selbst, wann die Kontrolle an das Betriebssystem zurückgegeben wird → meist nach dessen Beendigung
  - Preemptive: Betriebssystem steuert, wann die Umschaltung zu einem anderen Prozess erfolgt → Prozesse können an beliebiger Stelle unterbrochen und fortgesetzt werden

Eine der wichtigsten Aufgaben des Process Managers ist die Erstellung eines Ablaufplans der aktiven Prozesse. Diese Aufgabe wird vom Scheduler übernommen, der darüber entscheidet, wie die CPU-Zuteilung zu den einzelnen Prozessen erfolgen soll. Dabei wird jedem Prozess ein Zeitintervall (Zeitscheibe) zugeordnet, in dem er die CPU benutzen kann. Je nachdem wie die Rückgabe der Zuteilung an das Betriebssystem erfolgt, unterscheidet man beim Scheduling zwei unterschiedliche Strategien.

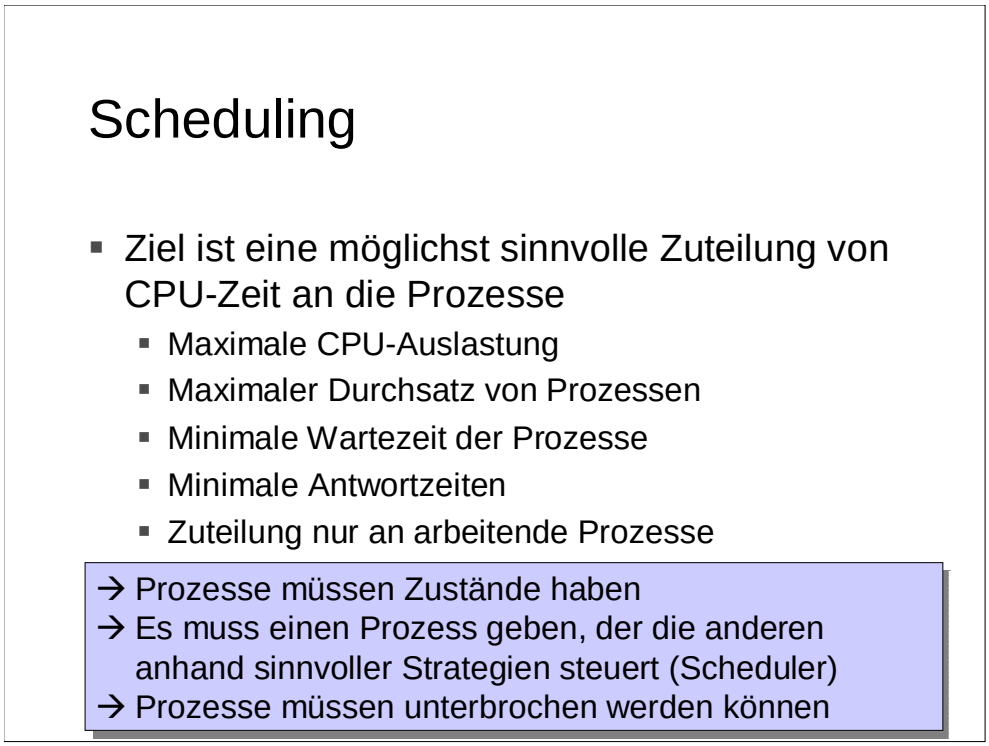
Nonpreemptive Scheduling

Bei dieser Strategie ist der Prozess selbst für die Rückgabe der Kontrolle an das Betriebssystem verantwortlich. Der Vorteil dieser Lösung liegt in der Entlastung des Process Managers, da dieser nicht ständig die Laufzeit der Prozesse überwachen muss. Der große Nachteil liegt aber darin, dass ein abgestürzter Prozess das Betriebssystem blockiert und insofern in einem Absturz des Betriebssystems resultiert. Weiters muss die Rückgabe in den Programmen selbst vorgesehen sein, da sonst die Rückgabe erst bei Beendigung des Prozesses erfolgt.

Preemptive Scheduling

Bei dieser Strategie wird die Umschaltung zwischen den Prozessen durch das Betriebssystem gesteuert. Dabei teilt das Betriebssystem den aktiven Prozessen Zeitscheiben zu, in denen der Prozess die CPU benutzen darf. Nach Ablauf der Zeitscheibe wird dem Prozess die CPU entzogen und dem nächsten eingeplanten Prozess zugeordnet. Darüber hinaus sorgt der Process Manager dafür, dass die Prozesse jeweils ihre korrekte Umgebung (Prozesskontext) vorfinden. Daraus resultiert, dass Prozesse an jeder beliebigen Stelle unterbrochen und wieder fortgesetzt werden können.

Scheduling



Wie zuvor erwähnt, ist das Ziel des Schedulers eine möglichst sinnvolle Zuteilung von CPU-Zeit an die einzelnen im System aktiven Prozesse zu finden. Diese Zuteilung ist aber in der Realität nicht einfach, da oft ein Kompromiss zwischen gegensätzlichen Zielen der einzelnen Prozesse gefunden werden muss. Der Scheduler muss beim Erstellen des Ausführungsplans auf einige Eckdaten von Prozessen Rücksicht nehmen und versucht, einen Ausführungsplan zu erstellen der folgenden Anforderungen gerecht wird:

- Maximale CPU-Auslastung: Der Ausführungsplan sollte die zur Verfügung stehende CPU-Zeit so optimal wie möglich ausnutzen. Prozesse, die auf externe Eingaben warten und insofern nicht bearbeitet werden können, sollten nicht in den Ausführungsplan aufgenommen werden.

- Maximaler Durchsatz von Prozessen: Definiert die Anzahl der Prozesse, die in einer Zeiteinheit bearbeitet werden können. Ein großer Durchsatz von Prozessen erhöht den Eindruck der Parallelität von Prozessen.

- Minimale Wartezeit der Prozesse: Die Wartezeit der eingeplanten Prozesse bis zur Aktivierung sollte so gering wie möglich sein. Lange Wartezeiten bewirken den Eindruck eines langsamen und überlasteten Systems.

- Minimale Antwortzeiten: Definiert die Zeit, die verstreicht bis ein Kommando vom Benutzer aktiv im Prozess umgesetzt wird. Lange Antwortzeiten bewirken den Eindruck von nicht reagierenden Programmen oder eines überlasteten Systems. Gerade bei interaktiven Programmen (Programmen mit Benutzerschnittstellen) sollten die Antwortzeiten so gering als möglich sein.

- Zuteilung nur an arbeitende Prozesse: Um das System nicht unnötig zu belasten und um den zuvor erwähnten Anforderungen gerecht zu werden, sollte der Scheduler nur solche Prozesse berücksichtigen, die wirklich in der Lage sind, Tätigkeiten durchzuführen. Prozesse, die auf das Freiwerden von Ressourcen, externe Ereignisse oder Eingaben warten sollten nicht in den Ausführungsplan aufgenommen werden.

Das Erstellen eines optimalen Ausführungsplans ist keine triviale Aufgabe, da in einem System oftmals viele Prozesse mit unterschiedlichen Anforderungen aktiv sind. So ist z.B. bei Prozessen mit hoher Benutzerinteraktion eine minimale Antwortzeit sehr wichtig, nicht aber für im Hintergrund laufende Batch-Jobs und Dienste. Solche Prozesstypen benötigen eher eine minimale Wartezeit.

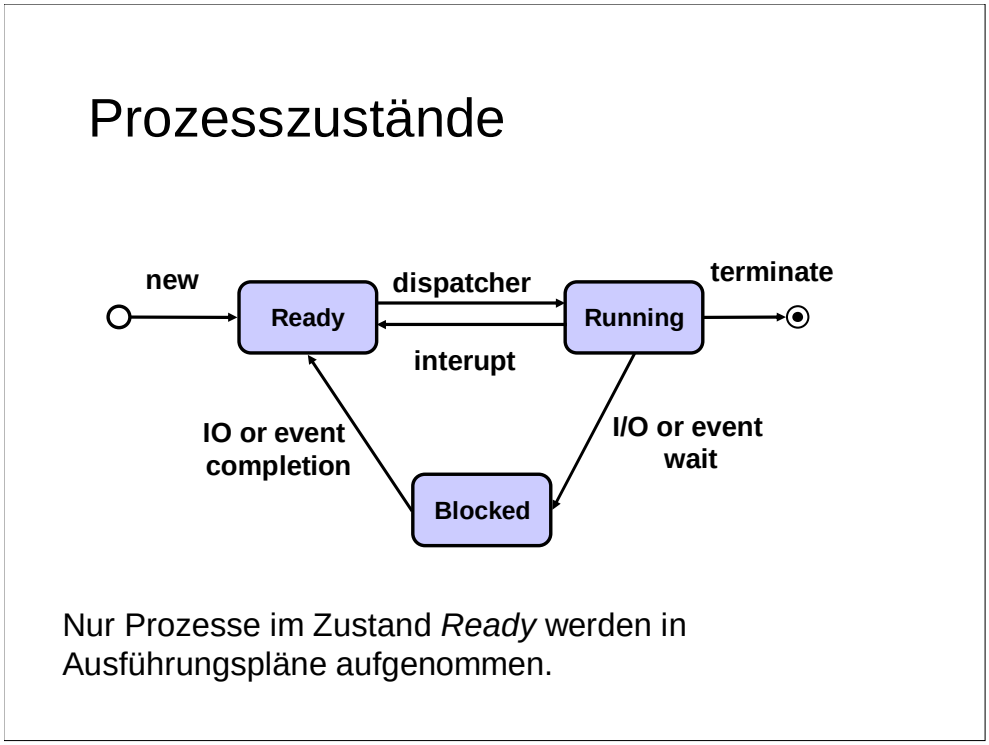
Aus den hier erwähnten Anforderungen an einen Ausführungsplan können folgende Erkenntnisse gewonnen werden:

1.) Prozesse müssen Zustände haben: Anhand der Zustände kann der Scheduler entscheiden, ob und wie ein Prozess in den Ausführungsplan eingeplant werden soll. Prozesse, die durch das Warten auf externe Ereignisse blockiert sind (Zustand *blocked*) sollen nicht in den Ausführungsplan aufgenommen werden.

2.) Es muss im System einen Prozess geben, der andere Prozesse steuert. Diesen Prozess haben wir bereits als Scheduler kennen gelernt. Allerdings benötigt dieser Prozess selbst CPU-Zeit und Ressourcen, die vom System zur Verfügung gestellt werden müssen.

3.) Prozesse müssen an jeder Stelle unterbrochen und wieder fortgesetzt werden können. Dieses Faktum ergibt sich aus der Tatsache, dass der Scheduler nach Ablauf der zugeordneten Zeitscheibe dem Prozess die CPU entzieht und einem anderen Prozess zuordnet. Ist der Prozess mit seiner Tätigkeit nicht fertig, so wird dieser wieder in den Ausführungsplan aufgenommen und zu einem späteren Zeitpunkt an genau der Stelle wo er unterbrochen wurde fortgesetzt.

Prozesszustände



Wie wir gerade gesehen haben, müssen Prozesse über Zustände verfügen. Tatsächlich verfügen Prozesse über mindestens drei verschiedene Zustände, nämlich *ready*, *running* und *blocked*, wobei nur Prozesse im Zustand *ready* in den Ausführungsplan aufgenommen werden. In weiterer Folge sollen nun die einzelnen Zustände näher erläutert werden.

Ready

Wird ein Prozess erzeugt, so befindet er sich automatisch im Zustand *ready*. Ein Prozess in diesem Zustand ist in der Lage, aktiv Tätigkeiten durchzuführen und kann somit vom Scheduler eingeplant werden.

Running

Sobald ein Prozess die CPU erhält und somit aktiv wird, wechselt er vom Zustand *ready* in den Zustand *running*. In diesem Zustand stehen dem Prozess alle von ihm benötigten Ressourcen zur Verfügung und wird von der CPU aktiv bearbeitet. Dieser Zustand kann auf mehrere Arten verlassen werden:

- Der Prozess ist fertig (letztes Statement abgearbeitet): In diesem Fall terminiert der Prozess und alle mit dem Prozess verbundenen Ressourcen werden vom System freigegeben.

- Der Prozess startet eine blockierende Anweisung: Blockierende Anweisungen erwarten externe Ereignisse, Eingaben oder warten auf das Freiwerden einer benötigten Ressource. Die Programmausführung kann erst dann fortgesetzt werden, wenn dieses Ereignis eintritt. Der Prozess

wechselt in den Zustand *blocked* und wird erst wieder eingeplant, wenn dieser aktiv weiterarbeiten kann (Ereignis ist eingetroffen).

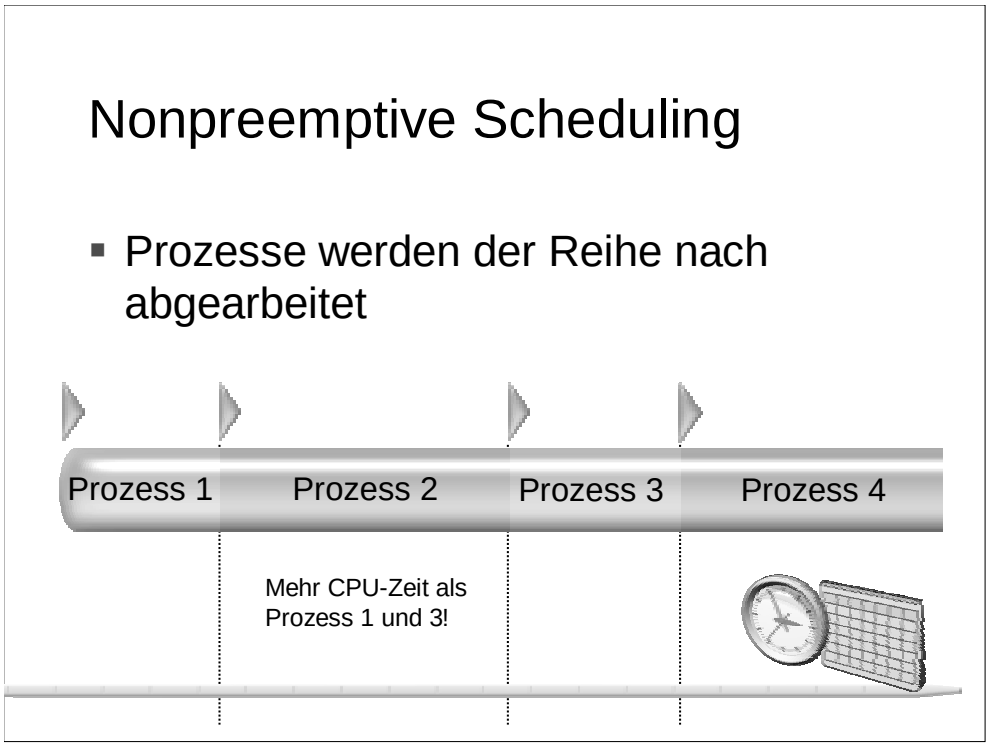
- Zeitscheibe ist abgelaufen: Der Prozess hat seine zugeteilte Zeit vollständig ausgenutzt und wird vom System unterbrochen. Da der Prozess in der Lage ist weiterzuarbeiten, wechselt er in den Zustand *ready* und wird vom Scheduler in den Ausführungsplan eingeplant.

Blocked

In diesem Zustand befinden sich alle Prozesse, die auf externe Ereignisse, die Freigabe einer benötigten Ressource oder Eingaben warten und deshalb nicht aktiv im System arbeiten können. Solche wartenden Prozesse werden nicht vom Scheduler in den Ausführungsplan aufgenommen. Tritt das erwartete Ereignis ein, so wechselt der Prozess vom Zustand *blocked* in den Zustand *ready* und wird wieder vom Scheduler in den Ausführungsplan eingeplant.

In weiterer Folge sollen nun die einzelnen Scheduling-Strategien näher betrachtet werden.

Nonpreemptive Scheduling



Beim nonpreemptive Scheduling sind die Prozesse jeweils selbst für die Rückgabe der Kontrolle an das Betriebssystem verantwortlich. Diese Rückgabe erfolgt im Normalfall nach der Abarbeitung des Prozesses. Um den Eindruck von Multitasking zu vermitteln, müssen Programme in mehrere kleinere Prozesse zerlegt werden, die dann jeweils komplett abgearbeitet werden. Da längere Prozesse die CPU auch länger benutzen dürfen, leidet bei dieser Strategie die minimale Wartezeit und minimale Antwortzeit. Dies kann etwas verbessert werden, indem der Scheduler kürzere Prozesse vor den längeren Prozessen einplant (siehe Abbildung 1). Allerdings wirkt sich diese Verbesserung nur zwischen den kürzeren Prozessen aus, lange Prozesse weisen nach wie vor eine lange Wartezeit und Antwortzeit auf.

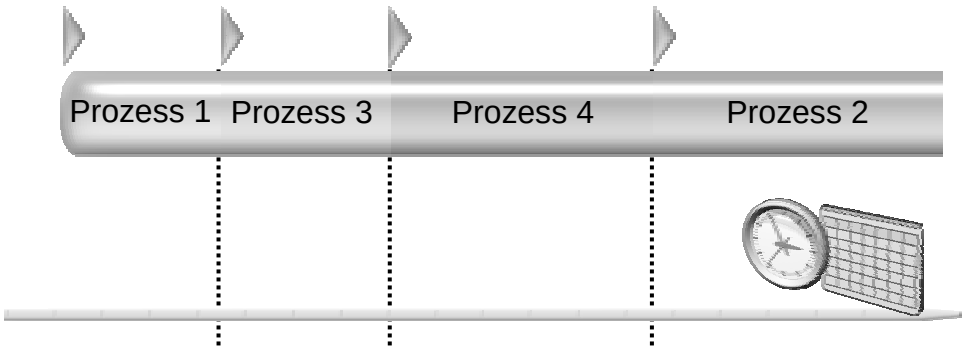


Abbildung 1: Optimierung: Shortest Job First

Preemptive Scheduling

Preemptive Scheduling

- Round Robin Scheduling
  - Jedem Prozess wird eine Zeitscheibe zugeordnet
  - Kann er innerhalb dieser Zeitscheibe seine Aufgabe nicht erledigen wird er verdrängt und erneut eingeplant
  - Führt der Prozess während Abarbeitung eine blockierende Aufgabe durch (z.B. Ein-/Ausgabe) wird dieser erst wieder eingeplant, wenn die entsprechende Tätigkeit durchgeführt werden kann (z.B. Eingabe erfolgt)

Beim preemptive Scheduling wird die CPU-Verfügbarkeit in Zeiteinheiten (den so genannten Zeitscheiben) aufgeteilt und den aktiven Prozessen zugeteilt. Der bekannteste Algorithmus hierfür ist das *Round Robin Scheduling*. Hier bekommt jeder Prozess eine Zeitscheibe zugeordnet, innerhalb dessen er die CPU aktiv benutzen kann. Kann der Prozess seine Tätigkeiten innerhalb der Zeitscheibe nicht erledigen, so wird er unterbrochen und von einem anderen Prozess verdrängt. Allerdings wird der Prozess erneut eingeplant, sodass er seine Tätigkeit zu einem späteren Zeitpunkt fortsetzen kann. Sobald ein Prozess eine blockierende Tätigkeit durchführt und die CPU nicht mehr aktiv benutzen kann, wird dieser verdrängt und erst dann wieder eingeplant, sobald die Tätigkeit fortgesetzt werden kann.

Da die Zeitscheiben der Prozesse gleich sind, hat das System eine vorhersehbare Wartezeit und Antwortzeit, sofern die Größe der Zeitscheibe und die damit Verbunden Anzahl von Context Switchs sinnvoll gewählt wurde.

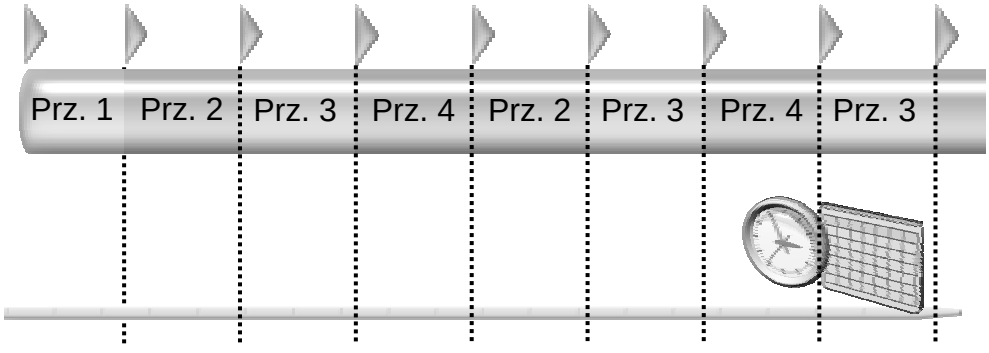


Abbildung 2: Round Robin Scheduling

Prioritätssteuerung

Preemptive Scheduling

- Optimierung: Prioritätssteuerung
  - Jeder Prozess enthält eine Priorität
    - Kann statisch vorgegeben sein
    - Kann dynamisch geändert werden
  - Laufender Prozess muss kleinere Priorität haben als neu hinzukommender Prozess
    - Prozess muss verdrängbar sein
  - Prozess mit niedriger Priorität wird von Prozess mit höherer Priorität verdrängt

Die Round Robin Strategie geht davon aus, dass im System alle Prozesse gleich wichtig sind. Oftmals besteht jedoch die Anforderung, dass Prozesse von unterschiedlichen Benutzern oder mit unterschiedlicher Aufgabe eine höhere Priorität und insofern über eine schnellere Antwortzeit und eine kürzere Wartezeit verfügen müssen als andere Prozesse.

Bei der Prioritätssteuerung wird jedem Prozess eine Priorität zugeordnet, die einerseits statisch vorgegeben sein kann (z.B. welcher Benutzer hat den Prozess gestartet) und / oder dynamisch geändert werden kann. Der Scheduler berücksichtigt bei der Vergabe der Zeitscheiben die Prioritäten der Prozesse und vergibt jeweils die CPU an den Prozess mit der höchsten Priorität.

Um zu vermeiden, dass ein hoch priorisierter Prozess die CPU permanent in Anspruch nimmt, vermindert der Scheduler die Priorität eines Prozesses in gewissen Zeitabständen. Sobald es im System einen Prozess mit höherer Priorität als dem Aktuellen, wird der aktuelle Prozess durch den höher Priorisierten verdrängt (Context Switch).

Multi Level Queue Scheduling

Multi Level Queue Scheduling

- Verschiedene Scheduling-Strategien kombiniert
  - Mehrere Warteschlangen werden in einen Ausführplan zusammengelegt
  - z.B. Interaktive Prozesse in Warteschlange mit hoher Priorität und Round Robin Scheduling, Batch-Prozesse in Warteschlange mit niedriger Priorität und der Reihe nach
  - Um *Verhungern* von Prozessen zu verhindern, wandern Prozesse zwischen Warteschlangen

Beim Multi Level Queue Scheduling verfügt das System über mehrere Warteschlangen, in die Prozesse eingeordnet werden. Die Warteschlangen selbst verfügen über eine unterschiedliche Priorität. Im Prinzip werden bei diesem Verfahren mehrere Scheduling Strategien kombiniert. Interaktive Prozesse könnten zum Beispiel in eine Warteschlange mit hoher Priorität eingeordnet werden und dann mittels Round Robin Scheduling abgearbeitet werden. Dies würde vernünftige Warte- und Antwortzeiten garantieren. Batch-Prozesse könnten in eine Warteschlange niedriger Priorität eingeordnet werden und anhand der Shortest Job First-Strategie abgearbeitet werden. In diesem Szenario könnte es aber sein, dass Batch Jobs nie an die Reihe kommen, da es immer interaktive Prozesse gibt die mit höherer Priorität laufen. Um ein solches Verhungern von Prozessen zu verhindern, könnte zusätzlich ein Mechanismus dafür sorgen, dass Prozesse anhand verschiedener Kriterien zwischen den Warteschlangen wandern können.

Realtime Scheduling

Realtime Scheduling

- Es gibt absolute Zeitpunkte (Deadlines)
  - Hard Real Time: Zeitpunkte müssen unbedingt eingehalten werden
  - Soft Real Time: Seltenes verfehlen der Zeitpunkte in einer Toleranzgrenze zulässig
- Programme werden in kleine Prozesse unterteilt
  - Die Ausführungszeit jedes dieser Prozesse ist dem System bekannt und insofern einplanbar
  - Scheduler berücksichtigt extern eintretende Ereignisse und sorgt dafür, dass diese von den richtigen Prozessen innerhalb der Deadlines bearbeitet werden

Die bis dato vorgestellten Strategien versuchen, die CPU so fair als möglich zwischen den Prozessen aufzuteilen. Allerdings können die Strategien keine Antwortzeiten von Prozessen garantieren und sind somit nicht für den Einsatz in Echtzeitsystemen geeignet. In solchen Systemen ist die Zeit eine absolut kritische Komponente und die darin verwendete Scheduling-Strategie muss Antwortzeiten von Prozessen garantieren können.

Zusätzlich unterscheidet man *Hard Real Time* Systeme, wo Zeitpunkte unbedingt eingehalten werden müssen und *Soft Real Time* Systeme, bei denen ein seltenes Verfehlen der Zeitpunkte innerhalb einer Toleranzgrenze zulässig ist.

Um ein solches System realisieren zu können, müssen Programme in kleine und überschaubare Prozesse unterteilt werden, wobei die Ausführungszeit dieser Prozesse dem System bekannt ist. Somit sind die Prozesse einplanbar, da das System weiß, wie lange die Abarbeitung der jeweiligen Prozesse dauert. Zusätzlich berücksichtigt der Scheduler das Eintreten externer Ereignisse und sorgt dafür, dass diese von den richtigen Prozessen innerhalb deren Deadlines bearbeitet werden können.

In der Praxis ist die Implementierung solcher Systeme oft problematisch aufgrund der langen Umschaltzeiten zwischen den Prozessen. Prozesse müssen insofern eigens gestaltet sein um Echtzeitfähigkeit zu unterstützen.

Der Context Switch

Context Switch

- Umschalten von einem Prozess zu einem anderen
  - Benötigt gewisse Zeit → Context Switch Time
- Gründe für Context Switch
  - Zeitscheibe des Prozesses ist abgelaufen
  - Aktiver Prozess startet blockierende Aufgabe
  - Aktiver Prozess ist fertig → terminiert
  - Prozess mit höherer Priorität ist bereit
  - Externes Ereignis (Unterbrechung) ist eingetreten

Sobald ein Prozess seine Zeitscheibe aufgebraucht hat, wird er normalerweise von einem anderen Prozess verdrängt. Diese Verdrängung wird *Context Switch* genannt. Dieser Context Switch benötigt Zeit, die so genannte *Context Switch Time*, die bei der verwendeten Scheduling Strategie berücksichtigt wird. Übersteigt die Context Switch Time eine kritische Grenze, so ist der verwendete Scheduling-Algorithmus ineffizient, da zu viel Zeit beim Context Switch verloren geht.

Ein Context Switch wird typischerweise durch folgende Ereignisse ausgelöst:

- Zeitscheibe eines Prozesses ist abgelaufen: Der Prozess wird durch einen anderen Prozess verdrängt.
- Aktiver Prozess startet blockierende Aufgabe: Der Prozess wartet auf ein externes Ereignis und kann deshalb nicht weiterarbeiten. Er wird durch einen anderen Prozess verdrängt.
- Aktiver Prozess ist fertig: Der Prozess terminiert und wird durch einen anderen Prozess ersetzt.
- Prozess mit höherer Priorität ist bereit: Der aktuelle Prozess wird durch einen höher priorisierten Prozess verdrängt.
- Externes Ereignis ist eingetreten: Der aktuelle Prozess muss unterbrochen werden um das externe Ereignis behandeln zu können.

Context Switch

1. Sicherung des aktuellen Prozesskontexts
   - Register, Programmzähler, OS-Daten
2. Aktualisierung der Prozessinformation
3. Einordnung in entsprechende Warteschlange des Schedulers
4. Scheduler wählt nächsten aktiven Prozess aus
5. Laden und aktualisieren der Prozessinformation des nächsten aktiven Prozesses
6. Laden und aktualisieren der Datenstrukturen
7. Laden des Kontexts des ausgewählten Prozesses

Im Folgenden soll nun der Ablauf und die Tätigkeiten beschrieben werden, die bei einem Context Switch erfolgen.

Im ersten Schritt wird der aktuelle Prozesskontext gesichert. Dazu zählen die aktuell verwendeten Register, der Programmzähler (*Instruction Pointer*) und andere Kenndaten des Prozesses. Diese Kenndaten und die Prozessinformation werden in einem weiteren Schritt durch den Process Manager aktualisiert. Wird ein Prozess verdrängt, der in den Zustand *ready* wechselt, so erfolgt nun bereits eine Einordnung in die entsprechende Warteschlange des Schedulers (Einplanung in Ablaufplan).

Der Scheduler wählt in einem vierten Schritt nun den nächsten Prozess aus, der aktiv geschaltet werden soll. Die für diesen Prozess benötigten Prozess-informationen werden vom Process Manager geladen und aktualisiert. Im letzten Schritt wird nun der Kontext des neuen Prozesses geladen und bereitgestellt.

Prozessinformation

Prozessinformation

- Prozesse sind in Betriebssystemen durch eine Datenstruktur implementiert
- Jeder Prozess besitzt Prozessinformation (Process Control Block)
  - Daten zur Prozessverwaltung
  - Wird beim Erzeugen des Prozesses erstellt
  - Prozess- und Laufzeitinformationen

Wie zuvor erwähnt, muss das Betriebssystem über jeden Prozess Daten zur Verfügung halten und diese laufend aktualisieren. Dazu besitzt jeder Prozess einen so genannten *Process Control Block*, wo Prozessinformationen zur Prozessverwaltung gehalten werden. Dieser Process Control Block wird beim Erzeugen des Prozesses erstellt und vom Process Manager laufend aktualisiert. Der Process Control Block enthält folgende Informationen:

- Identifikationsnummer (Prozess ID)
- Prozesszustand
- Instruction Pointer (nächste auszuführende Anweisung)
- Registerinhalte
- Zugeordnete Speicherbereiche
- Liste von geöffneten Dateien
- Priorität des Prozesses
- Status der Ein-/Ausgabegeräte, die von Prozess benötigt werden

Ereignisbehandlung

Ereignisbehandlung

- Scheduler und Betriebssystem wird größtenteils durch Ereignisse beeinflusst
  - Timer: Zeitscheibe eines aktiven Prozesses ist abgelaufen
  - Blockierende Anweisung: Systemaufrufe werden mittels Unterbrechungen realisiert
  - Daten für blockierende Anweisung sind verfügbar
- Zwei Möglichkeiten auf Ereignisse zu reagieren
  - Polling (In definierten Zeitintervallen wird geprüft, ob Ereignis eingetreten ist → busy waiting)
  - Unterbrechungen (aktueller Prozess wird unterbrochen und Ereignis wird in Routine behandelt → Behandlung erfolgt nur dann, wenn Ereignis wirklich eingetreten ist!)

Um die in diesem Kapitel erwähnten Aufgaben bewerkstelligen zu können, muss das Betriebssystem, dessen Prozesse und insbesondere der Process Manager die Möglichkeit haben, auf verschiedenste Ereignisse zu reagieren. Tatsächlich wird die Arbeit des Betriebssystems durch das Auftreten von Ereignissen gesteuert und beeinflusst. Beispiele hierfür sind zeitliche Ereignisse wie z.B. das Ablaufen einer Zeitscheibe eines Prozesses, der Aufruf einer blockierenden Anweisung und das Eintreffen der Daten einer blockierenden Anweisung.

Das Betriebssystem muss auf solche Ereignisse reagieren und diese behandeln. Teilweise löst das Betriebssystem selbst Ereignisse auf um dringende Arbeiten zu erledigen. Dabei wird ein aktuell ausgeführter Prozess unterbrochen und eine so genannte Ereignisbehandlungsroutine ausgeführt. Nach der Behandlung des Ereignisses wird mit der Bearbeitung des Prozesses fortgesetzt.

Es gibt zwei unterschiedliche Strategien, wie das Betriebssystem auf mögliche Ereignisse reagiert und diese behandelt.

Polling

Beim Polling wird in definierten Zeitintervallen regelmäßig überprüft, ob ein Ereignis aufgetreten wird. Da eine Überprüfung auch dann erfolgt, wenn kein Ereignis aufgetreten ist, nennt man dieses Verfahren auch *busy waiting*. Polling hat die Eigenschaft, dass die CPU unnötig belastet wird, da immer auf alle möglichen Ereignisse geprüft wird, auch wenn diese noch nicht eingetreten sind.

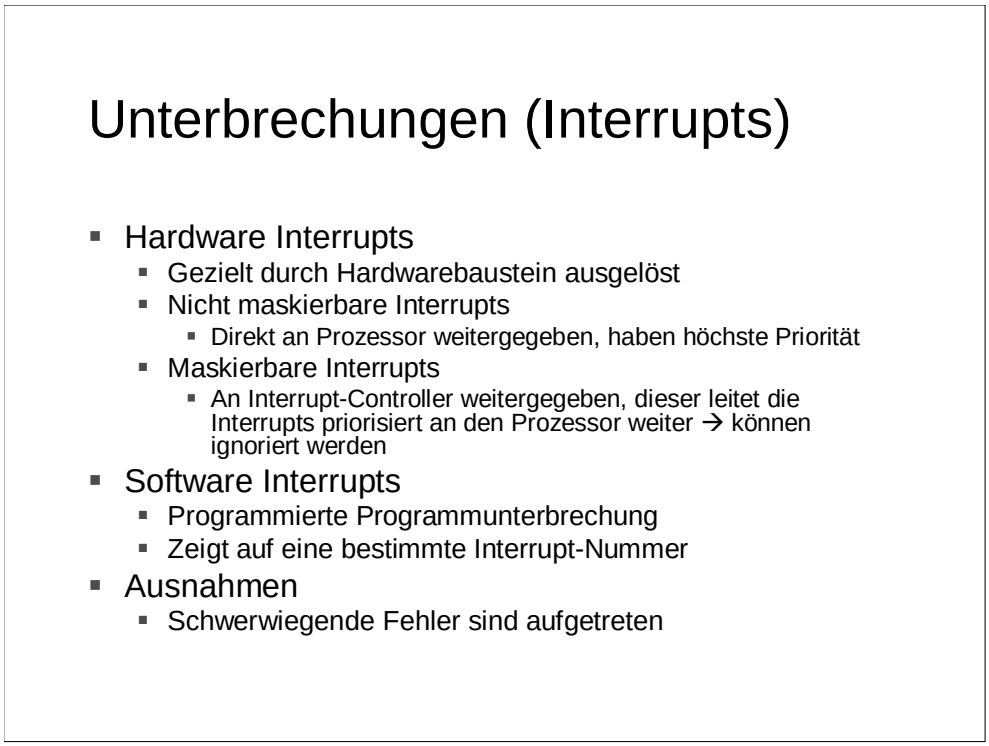
Unterbrechungen (Interrupts)

Bei dieser Strategie unterbricht ein eintretendes Ereignis die Ausführung des aktuellen Prozesses und bewirkt die sofortige Bearbeitung durch eine entsprechende Routine (Ereignisbehandlungsroutine). Nach Bearbeitung des Ereignisses wird mit der normalen Prozessausführung fortgesetzt.

Diese Strategie hat den Vorteil, dass nur dann eine Behandlung erfolgt wenn auch wirklich ein Ereignis eingetreten ist. Allerdings muss das Betriebssystem Mechanismen bereitstellen, dass Ereignisbehandlungsroutinen erstellt und gewartet werden können.

Da aus Gründen von Effizienz fast ausschließlich Unterbrechungen eingesetzt werden, wird diese Strategie in weiterer Folge ausführlicher behandelt.

Unterbrechungen



Unterbrechungen werden in zwei Kategorien eingeteilt, je nach dem von wo die Unterbrechungsanforderung ausgelöst wurde.

Hardware Interrupts

In diese Kategorie fallen alle Unterbrechungen die gezielt durch Hardware ausgelöst wurden. Je nach Aussagekraft kann die Abarbeitung der Unterbrechung beeinflusst werden.

- Nicht maskierte Interrupts: Eine solche Unterbrechungsanforderung hat sehr hohe Priorität und muss sofort von der CPU bearbeitet werden. Deshalb werden nicht maskierte Interrupts sofort an den Prozessor übermittelt. Der Prozessor arbeitet den gerade ausgeübten Befehl ab und führt unmittelbar anschließend einen Interrupt 2 durch. Solche Interrupts kennzeichnen schwerwiegende Hardwarefehler und werden auch nur in Ausnahmefällen ausgelöst.

- Maskierte Interrupts: Unterbrechungsanforderungen dieser Kategorie werden nicht direkt an die CPU übermittelt sondern an den Interrupt-Controller. Dieser verwaltet mehrere Interrupt-Anforderungen und gibt sie geordnet nach Priorität an den Prozessor weiter. Zudem besteht die Möglichkeit, Interrupts dieser Kategorie zu ignorieren.

Interrupt Behandlungsroutinen

- Interrupt Vektorentabelle mit Startadressen der Behandlungsroutinen
- Behandlungsroutinen im Betriebssystem implementiert, können aber auch von Benutzer implementiert werden (z.B. Embedded Systems)

Nr	Address	Description
00	000-003	Division / Zero
…	…	…
08	020-023	(IRQ 0) Timer
09	024-027	(IRQ 1) Keyboard
0A	028-02B	(IRQ 2)
0B	02C-02F	(IRQ 3) COM2
0C	030-033	(IRQ 4) COM1
…	…	…
16	058-05B	BIOS: Keyboard
…	…	…
70	1C0-1C3	Realtime Clock
…	…	…

Software Interrupts

Ein Software Interrupt ist eine programmierte Programmunterbrechung, die mit Hilfe von Systemaufrufen ausgelöst werden können. Dazu muss die Nummer des benötigten Interrupts bekannt sein, die als Hexadezimaladresse im Programm aufgerufen werden. Das System führt eine so genannte Interrupt-Vektor-Tabelle, die jeweils zu einer Interrupt-Nummer die Adresse eines Interrupt-Behandlungs-programms (Behandlungsroutine) bereitstellt. Somit kann das Betriebssystem Dienste anbieten, die dann von Benutzerprogrammen verwendet und jederzeit abrufbar sind.

Die Behandlungsroutinen sind teilweise durch das Betriebssystem vorgegeben und ermöglichen die Benutzung von Diensten im Betriebssystem bzw. den Zugriff auf Hardware. Allerdings können die Routinen auch von Benutzern implementiert werden, z.B. zum Anbieten von Routinen in eingebetteten Systemen.

Systemaufrufe

Systemaufrufe

- Funktionsaufruf zur Dienstanforderung
  - Benutzerprogramme dürfen nicht direkt auf Hardware zugreifen
  - Kernel wechselt in priviligierten Prozessormodus und kann auf Hardware zugreifen
  - Aufruf erfolgt mittels Software Interrupts
- Systemaufruf funktioniert wie der Aufruf einer Funktion
  - Parameter, Rückgabewerte
  - Prozess blockiert bis Aufruf beendet ist

Wie bereits in Kapitel 1 besprochen, laufen Benutzerprogramme in einer virtualisierten Umgebung mit eingeschränkten Rechten (*user mode*) und können nicht direkt auf die Hardware zugreifen. Um trotzdem in der Lage zu sein mit der Hardware zu kommunizieren, müssen diese Dienste des Betriebssystems in Anspruch nehmen, die im privilegierten Modus (*kernel mode*) laufen und direkt auf die Hardware zugreifen können. Die dazu benötigte Schnittstelle wird durch die Systemaufrufschnittstelle zur Verfügung gestellt.

Die Systemaufrufschnittstelle stellt eine Reihe von Funktionen zur Dienstanforderung zur Verfügung. Sobald eine solche Funktion aufgerufen wird, erfolgt ein Wechsel in den privilegierten Modus und ein Hardwarezugriff ist möglich. In der Praxis erfolgt dieser Aufruf durch die Anforderung eines bestimmten Software Interrrupts in einem Benutzerprogramm. Wie zuvor besprochen kann somit das Betriebssystem Funktionalität in der Interrupt-Vektortabelle bereitstellen, die so über Benutzerprogramme jederzeit aufgerufen werden können. Ein Aufruf bewirkt die Unterbrechung des Benutzerprogramms und die Ausführung der Interrupt-Behandlungsroutine, die selbst im privilegierten Modus ausgeführt wird.

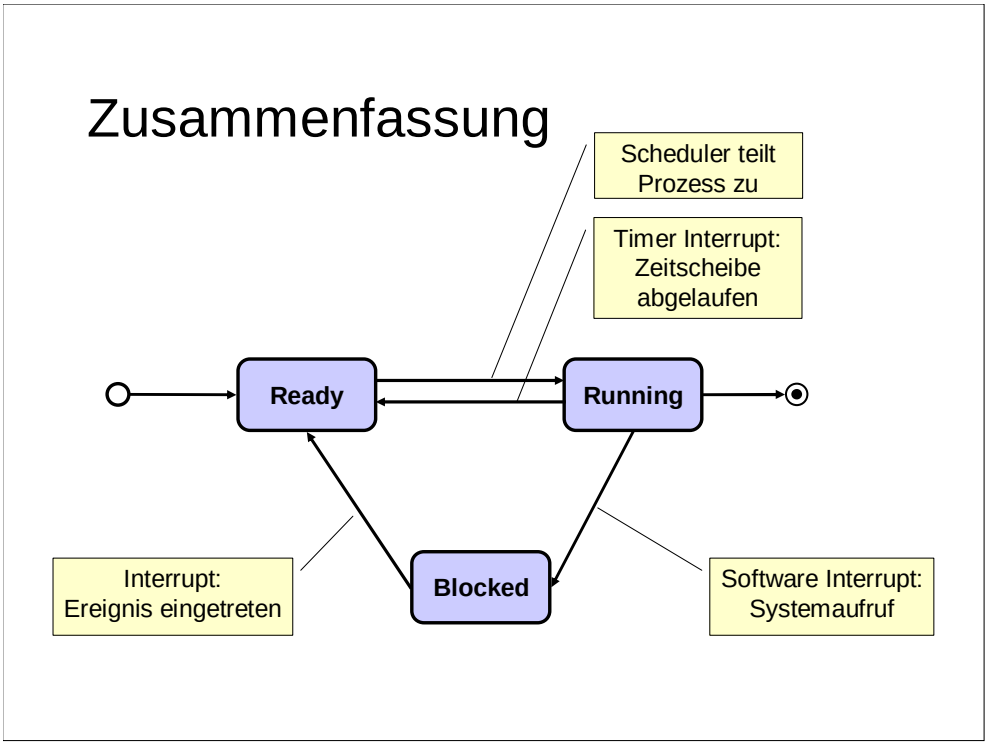
! Systemaufrufe sind immer blockierende Anweisungen, d.h. der aufrufende Prozess blockiert solange, bis der Systemaufruf abgeschlossen ist. Dies ist insofern logisch, da Systemaufrufe Interrupts auslösen und das Benutzerprogramm so lange unterbrochen wird, bis die Routine zurückkehrt und somit der Interrupt behandelt wurde.

Systemaufrufe funktionieren wie der Aufruf einer Funktion bei einer Programmiersprache. Diese Funktionen erwarten bestimmte Parameter und liefern Rückgabewerte. Meist müssen auch Buffer zur Übergabe größerer Datenmengen

bereitgestellt werden, die nach Aufruf der Funktion wieder freigegeben werden müssen.

Systemaufrufe sind meist an eine bestimmte Programmiersprache gebunden. In UNIX und Linux sind die Systemaufrufe an die Programmiersprache ‚C' gebunden, bei Windows bis zur Version „Windows Server 2003" sind Systemaufrufe an die Sprache „C++" gebunden. Einen interessanten Ansatz findet man bei Windows Vista, wo die Systemaufrufe bereits an die Programmierplattform „.NET" gebunden sind („WinFX"). Diese Plattform unterstützt mehrere Sprachen (die so genannten .NET-Sprachen) und läuft in einer vom System geprüften Umgebung (*managed environment*). In einem solchen Umfeld gibt es jemanden, der den auszuführenden Code analysiert und dessen korrekte Ausführung überwacht.

Zusammenfassung



Zum Schluss soll das in diesem Kapitel gelernte noch mal anhand eines Beispiels verdeutlicht werden. Dabei wird der Lebenszyklus eines Prozesses veranschaulicht und die bei der Ausführung beteiligten Systemkomponenten.

In einem ersten Schritt wird der Prozess erstellt. Bei der Erstellung des Prozesses wird vom Process Manager zum einen ein Process Control Block mit den Laufzeitinformationen des Prozesses angelegt, zum anderen sorgt der Process Manager, dass der Prozess sein nötiges Umfeld (Process Context, virtual Memory, etc.) vorfindet. Der Prozess wechselt nach erfolgreicher Erstellung in den Zustand *ready* und wartet darauf, dass der Scheduler ihm eine Zeitscheibe zur Ausführung zuordnet.

Der Scheduler erstellt einen Ausführungsplan und berücksichtigt den in Schritt 1 erstellten Prozess, da dieser im Zustand *ready* ist. Sobald der Prozess an der Reihe ist, wird durch den Process Manager ein entsprechender Context Switch durchgeführt, sodass der Prozess sein richtiges Umfeld vorfindet (Einlagern von Speicherseiten, aktualisierter Prozesskontext, etc.). Der Prozess wechselt in den Zustand *running* und führt aktiv Anweisungen durch.

Sobald die Zeitscheibe des Prozesses aufgebraucht wird, wird vom Timer ein Interrupt ausgelöst woraufhin der Process Manager einen Context Switch durchführt und den aktiven Prozess durch einen anderen Prozess verdrängt. Da der Prozess gerade aktiv gearbeitet hat, wechselt dieser wieder in den Zustand *ready* und erwartet erneut auf seine Zuteilung. Diese Zuteilung erfolgt wie bereits zuvor gezeigt.

Wir gehen nun davon aus, dass unser Prozess wiederum eine Zeitscheibe erhalten hat und gerade aktiv eine Anweisung ausführt (im Zustand *running*). Der Prozess möchte nun über die Systemaufrufschnittstelle eine Datei lesen, die allerdings gerade von einem anderen Prozess geschrieben wird. Wie in diesem Kapitel gezeigt, erfolgt ein solcher Aufruf über die Systemaufrufschnittstelle durch das Auslösen eines Softwareinterrupts. Da alle diese Aufrufe blockierend sind, wird der Prozess in den Zustand *blocked* überführt und durch einen anderen Prozess verdrängt. Durch einen Interrupt wird auch die entsprechende Funktion ausgeführt, allerdings weiß der Process Manager durch Rücksprache mit dem File System Manager das die angeforderte Datei gerade durch einen anderen Prozess blockiert ist. Der Prozess bleibt insofern solange im Zustand *blocked* bis die Datei wieder zur Verfügung steht und wird deshalb nicht vom Scheduler eingeplant.

Irgendwann ist nun die Datei wieder verfügbar. Da das System weiß, dass der Prozess auf dieses Ereignis wartet, löst das System einen Interrupt aus, der unseren Prozess wieder vom Zustand *blocked* in den Zustand *ready* überführt. Somit plant der Scheduler den Prozess wieder in den Ausführungsplan ein. Sobald dieser an der Reihe ist, kann der Prozess nun seine Ausführung fortsetzen.

Irgendwann hat der Prozess seine Tätigkeiten vollendet und seine letzte Anweisung erfolgreich durchgeführt. In diesem Falle terminiert der Prozess woraufhin der Process Manager in Kombination mit den anderen Betriebssystemkomponenten die vom Prozess belegten Ressourcen bereinigt und freigibt.

Kapitel

3

Der Memory Manager

Dieses Kapitel erläutert den Memory Manager, dessen Funktion und Aufgaben im Detail.

Jedes Programm sowie das Betriebssystem benötigen Speicher, indem Daten und Informationen abgelegt und verwaltet werden können. Da die Datenmenge mitunter sehr groß werden kann und ein sehr dynamisches Verhalten hat, bedarf es einer eigenen Verwaltungseinheit im Betriebssystem, die den zur Verfügung stehenden Speicher verwaltet und den Prozessen zuordnet. Darüber hinaus wird öfters mehr Speicher benötigt als real zur Verfügung steht. Eine effiziente Nutzung von anderen Speichermedien (z.B. der Festplatte) kann zur Erweiterung des Hauptspeichers herangezogen werden. Dabei sind allerdings unterschiedliche Strategien nötig, um den Speicher optimal zu verwalten und trotzdem eine möglichst kurze Zugriffszeit (da andere Speichermedien meist längere Zugriffszeiten haben) zu ermöglichen. Diese Aufgaben übernimmt der *Memory Manager*.

In diesem Kapitel wird der Memory Manager, dessen Aufgaben und Funktion beschrieben. Dabei werden die unterschiedlichen Strategien und Algorithmen, die der Memory Manager zur effizienten Verwaltung des Hauptspeichers benötigt erläutert und vorgestellt. Besonderes Augenmerk wird auf das Konzept der virtuellen Adressierung von Prozessen und die Abbildung auf physikalische Adressen im Hauptspeicher gelegt. Dieses Konzept kann durch den *Paging*-Mechanismus erreicht werden, mit dem sich dieses Kapitel hauptsächlich beschäftigt.

Lehrinhalte und -ziele

Dieses Kapitel beschäftigt sich mit dem Memory Manager, dessen Aufgaben und Funktionalität. Besonders wird das Konzept der virtuellen Adressierung von Prozessen und deren Abbildung auf den Hauptspeicher gezeigt.

Sie sollten nach diesem Kapitel über die Aufgaben des Memory Managers und deren Umsetzung in Betriebssystemen bescheid wissen. Darüber hinaus sollten Sie wissen, wie die virtuelle Adressierung in Prozessen funktioniert sowie deren Abbildung auf den Hauptspeicher erfolgt. Darüber hinaus sollten Sie die Begriffe *Paging* und *Swapping* kennen und erklären können.

Aufgaben des Memory Managers

Memory Manager

- Verwaltung der Speicherhierarchie
  - Cache
  - RAM
  - Festspeicher

Größe Kosten, Geschwindigkeit

- Aufgaben
  - Verwaltung des Speichers (Liste mit freien, benutzten Speicherbereichen)
  - Stellt Prozessen Speicherbereiche zur Verfügung und gibt diese nach Beendigung wieder frei
  - Erweiterung des Hauptspeichers durch Auslagerung von Speicherbereichen auf die Festplatte (swapping)

Die Aufgabe des Memory Managers ist die Verwaltung des im Computer vorhandenen Speichers. Dabei werden die unterschiedlichen Speicherarten und deren Eigenschaften (Dauerhaftigkeit, Geschwindigkeit, Größe) berücksichtigt.

Hauptsächlich sorgt der Memory Manager dafür, dass der virtuelle Speicherbereich der jedem Prozess zugesichert ist, real zur Verfügung steht. In der Realität ist der virtuelle Speicherbereich meist größer als der physisch vorhandene Hauptspeicher. Deswegen ist der Memory Manager in der Lage, den Hauptspeicher durch Verwendung eines anderen Hintergrundspeichers (z.B. der Festplatte) zu erweitern. Die Kernproblematik dabei liegt in der effizienten Nutzung der im System vorhandenen unterschiedlichen Speichertypen, da sich die zur Verfügung stehenden Speicher stark voneinander in Hinblick auf Größe, Kosten und Geschwindigkeit unterscheiden. So ist der schnellste zur Verfügung stehende Speicher (Cache) meist sehr klein und sehr teuer, der größte zur Verfügung stehende Speicher (Festplatte) meist sehr groß und relativ günstig, dafür sehr langsam.

Zu den Aufgaben des Memory Managers gehört die Verwaltung des physikalischen Speichers. Dazu führt er eine Liste mit freien und belegten Speicherbereichen. Fordern Prozesse Speicherbereiche an, so stellt der Memory Manager diese zur Verfügung und gibt diese nach der Beendigung des Prozesses wieder frei. Da Prozesse auf Grund der ihnen zur Verfügung stehenden virtuellen Speicherbereiche meist mehr Speicher anfordern als real zur Verfügung steht, kann der Memory Manager auf Hintergrundspeicher wie z.B. die Festplatte zurückgreifen und somit den realen Hauptspeicher erweitern.

Virtueller Adressraum

Virtueller Adressraum

- Speicheradressierung erfolgt mit virtuellen Adressen
  - Abbildung in physikalische Adressen durch Memory Management Unit
  - Prozess kann mehr Speicher adressieren als physikalisch zur Verfügung steht
- Betriebssystem stellt jedem Prozess eigenen virtuellen Adressraum zur Verfügung → Benutzeradressraum
  - Abschottung von anderen Prozessen
- Betriebssystem stellt für alle Prozesse einen gemeinsamen Kerneladressraum zur Verfügung
  - HAL, Treiber, Prozessseitentabellen, Dateisystem-Cache, …

In modernen Betriebssystemen erhält jeder Prozess einen eigenen Adressraum, der von den anderen Prozessen abgeschottet ist. Dieser Adressraum scheint für den Prozess ein linearer und zusammenhängender Speicherbereich zu sein. Allerdings ist dieser Adressraum rein virtuell und wird im Hintergrund vom Betriebssystem auf mehrere, eventuell verteilte Hardwareadressen abgebildet. Diese Umsetzung der virtuellen Adressen auf die physikalischen Adressen wird durch die *Memory Management Unit* durchgeführt. Zudem wird die Größe des virtuellen Speichers durch die Adressbreite vorgegeben und kann viel größer sein als der real vorhandene Hauptspeicher. Prozesse können somit auch Adressen verwenden, die physikalisch auf dem Rechner nicht existieren.

Der Arbeitsspeicher selbst ist in so genannte *Speicherseiten* aufgeteilt, deren mögliche Größen durch die Hardware vorgegeben sind. Bei einer 32-Bit Architektur weisen die Adressen eine Länge von 32-Bit auf, wodurch sich 4 KB-Speicherseiten ergeben (32/8 = 4). Insgesamt kann so das Betriebssystem einen Adressraum von 4 GB (2^{32} Byte) adressieren. Jede virtuelle Adresse kann dann auf eine solche Speicherseite abgebildet werden. Greift ein Prozess auf eine Adresse zu, der keine Speicherseite zugeordnet ist, so muss das Betriebssystem durch die Memory Management Unit eine solche Speicherseite bereitstellen und der virtuellen Adresse zuordnen. Steht keine freie Speicherseite mehr zur Verfügung, so muss eine solche frei gemacht werden, wobei der Inhalt einer bestehenden Speicherseite auf die Festplatte ausgelagert wird. Dieser Vorgang wird auch *paging* genannt und wird später in diesem Kapitel im Detail erklärt.

Die Vorteile der virtuellen Speicherverwaltung ergeben sich aus der Möglichkeit, Prozessen mehr Speicher zur Verfügung zu stellen als real im System vorhanden

ist. Zum anderen erlaubt diese Technik die Implementierung von Speicherschutzmechanismen, indem die Speicherbereiche der einzelnen Prozesse voneinander komplett abgeschottet werden. Es ist somit nicht mehr möglich, dass ein Prozess irrtümlich (oder bösartiger weise) in den Speicherbereich eines anderen Prozesses schreibt. Den virtuellen Speicherbereich eines Prozesses nennt man auch *Benutzeradressraum*.

Das Betriebssystem benötigt jedoch auch einen Speicherbereich, der allen Prozessen zur Verfügung steht. In diesem Speicherbereich befindet sich das eigentliche Betriebssystem, dessen Komponenten und Dienste, die über die Systemaufrufschnittstelle von den Benutzerprogrammen genutzt werden können. Dieser Speicherbereich wird *Kerneladressraum* genannt und kann über die Systemaufrufschnittstelle gemeinsam genutzt werden.

Aufteilung der Adressräume

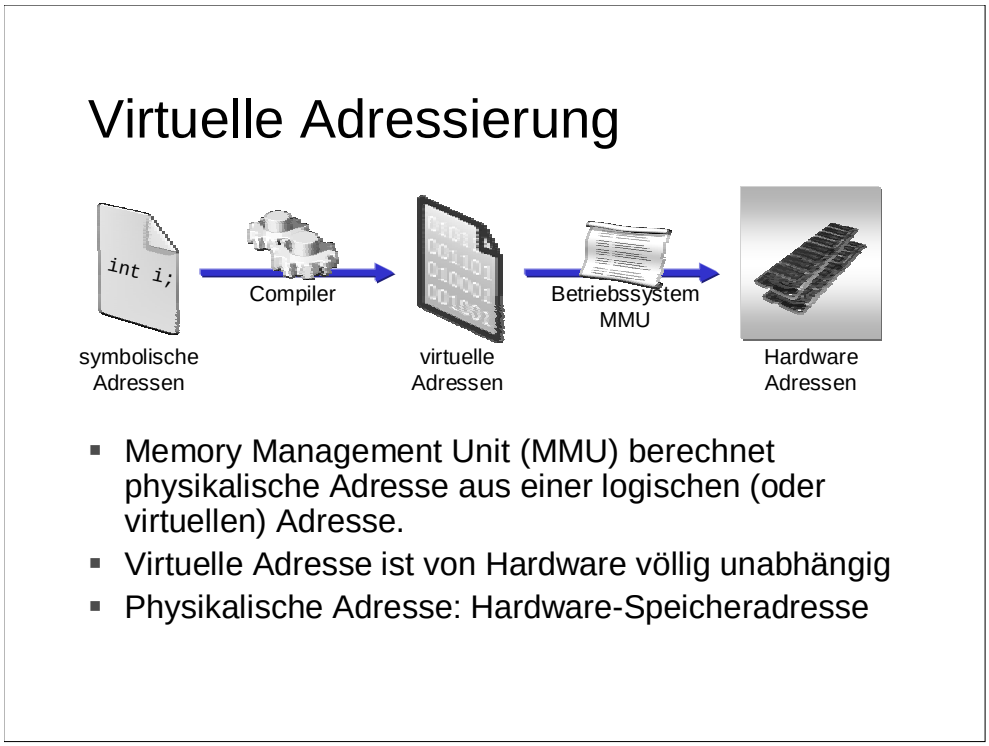
Wie zuvor erwähnt, können 32-Bit Betriebssysteme insgesamt 4 GB an Speicher adressieren. Dieser Speicher ist weiters in den Benutzeradressraum (Programme) und in den Kerneladressraum (Kernel) unterteilt. Die folgende Tabelle zeigt die Aufteilung bei den heute gängigsten Betriebssystemen.

Betriebssystem	Benutzerraum	Kernelraum
Windows	2 GB	2 GB
Linux	3 GB	1 GB

ⓘ Windows erlaubt durch Angabe eines Bootparameters in der Konfigurationsdatei „boot.ini“ die Änderung des Adressraums auf 3 GB Benutzer- und 1 GB Kerneladressraum. Dazu muss in der Konfigurationsdatei einfach beim Booteintrag der Parameter „/3GB“ angegeben werden.

ⓘ Neuere Hardware und Betriebssysteme unterstützen allerdings bereits 64 Bit Adresslängen. Aus einer solchen Adressbreite können nun 2^{64} Bit adressiert werden wodurch insgesamt 16 TB adressiert werden können.

Virtuelle Adressierung



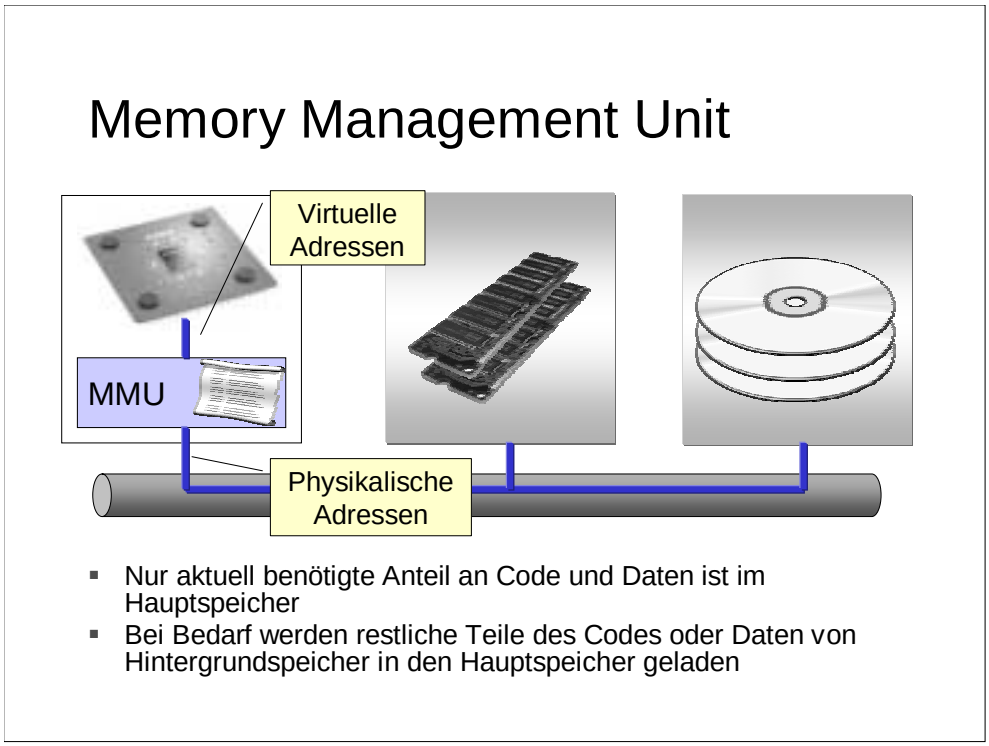
Im Folgenden sollen nun die unterschiedlichen Adressarten und deren Abbildung auf die Hardwareadressen durch das Betriebssystem und die Hardware veranschaulicht werden.

Adressarten

Bei der Erstellung eines Programms verwendet der Programmierer ausschließlich *symbolische Adressen* in Form von Variablen. Bei der Definition einer solchen Variablen wird für das restliche Programm eine symbolische Adresse mit dem Namen der Variablen erzeugt, wobei der Zugriff auf die darin gespeicherte Information immer über den Variablennamen erfolgt.

Beim Kompilieren werden die symbolischen Adressen auf *virtuelle Adressen* abgebildet. Diese virtuellen Adressen sind bereits Speicherbereiche im Benutzeradressraum, die jeweils die Variableninformation aufnehmen können. Allerdings sind diese Adressen nur virtuell und von der Hardware völlig unabhängig. Deshalb muss durch das Betriebssystem und die Memory Management Unit die virtuellen Adressen auf *physikalische Adressen* abgebildet werden. Dies erfolgt bei der Ausführung des Programms durch den Memory Manager. Die physikalischen Adressen sind wirklich im System vorhandene Speicherzellen in Form von Hardware. Das Programm selbst arbeitet aber immer nur mit den virtuellen Adressen.

Die Memory Management Unit



Die Memory Management Unit (MMU) ist ein Teil des Mirkoprozessors, dessen Aufgabe in der Übersetzung der virtuellen Adressen von Programmen in physikalische Adressen des Hauptspeichers besteht. Damit wird ein Zugriff auf den gesamten virtuellen Adressraum, den ein Betriebssystem zur Verfügung stellt, ermöglicht.

Funktionsweise der MMU

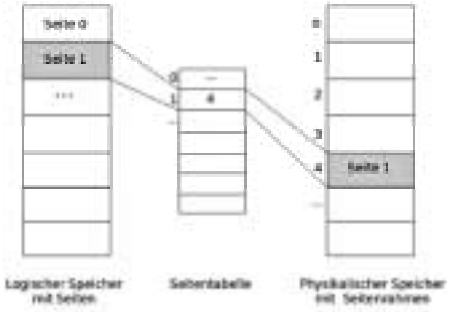
Jede von einem Prozess angeforderte virtuelle Adresse wird zunächst durch die MMU in eine physikalische Adresse umgesetzt. Bei dieser Umsetzung greift die MMU auf einen speziellen Cache-Speicher zurück (Translation Lookaside Buffer) welcher die letzten Adressübersetzungen zwischenspeichert und insofern oft genutzte Speicherbereiche schneller auffinden kann.

Da allerdings der physikalische Speicherbereich kleiner als der virtuelle Adressraum sein kann, ist nicht zwangsweise jeder virtuellen Adresse eine physikalische Adresse zugeordnet. Tatsächlich versucht das Betriebssystem, immer nur den aktuell benötigten Anteil an Code und Daten im Hauptspeicher zu halten und lagert den restlichen Teil auf einen Hintergrundspeicher (z.B. die Festplatte) aus. Adressiert ein Programm eine virtuelle Adresse, die keiner physikalischen Adresse zugeordnet ist, so tritt ein so genannter *Seitenfehler* auf, woraufhin das Betriebssystem den entsprechenden ausgelagerten Speicherbereich von der Festplatte laden kann. Dieser Vorgang obliegt vollständig dem Memory Manager und wird vor den Prozessen verborgen.

Paging

Paging

- Virtueller Adressraum wird in gleich große Stücke unterteilt → Pages
- Hardware Adressraum wird genauso unterteilt → Frames
- Pages und Frames sind gleich groß
- Seitentabelle ordnet Pages zu Frames zu → es existiert für jeden Prozess eine Seitentabelle (in der MMU)



Den Mechanismus zum Übersetzen von logischen Adressen in physikalische Adressen nennt man *Paging.* Da dies ein elementares Konzept in der Speicherverwaltung von modernen Betriebssystemen ist, soll diese Technik in weiterer Folge nun näher erläutert werden.

Zweck von Paging

Wie zuvor erwähnt wird jedem Prozess vom Betriebssystem ein Adressraum zugeordnet. Würde es sich hierbei um einen zusammenhängenden Speicherbereich handeln, so würden im Laufe der Zeit Lücken entstehen da Prozesse meistens unterschiedliche Speichermengen benötigen. Es würde somit zu einer Fragmentierung des Hauptspeichers kommen was wiederum mit erhöhten Zugriffszeiten einhergehen würde.

Paging vermeidet eine solche Fragmentierung, da zusammengehörige virtuelle Adressräume direkt auf die zugeordneten realen Speicheradressen zeigen.

Funktionsweise von Paging

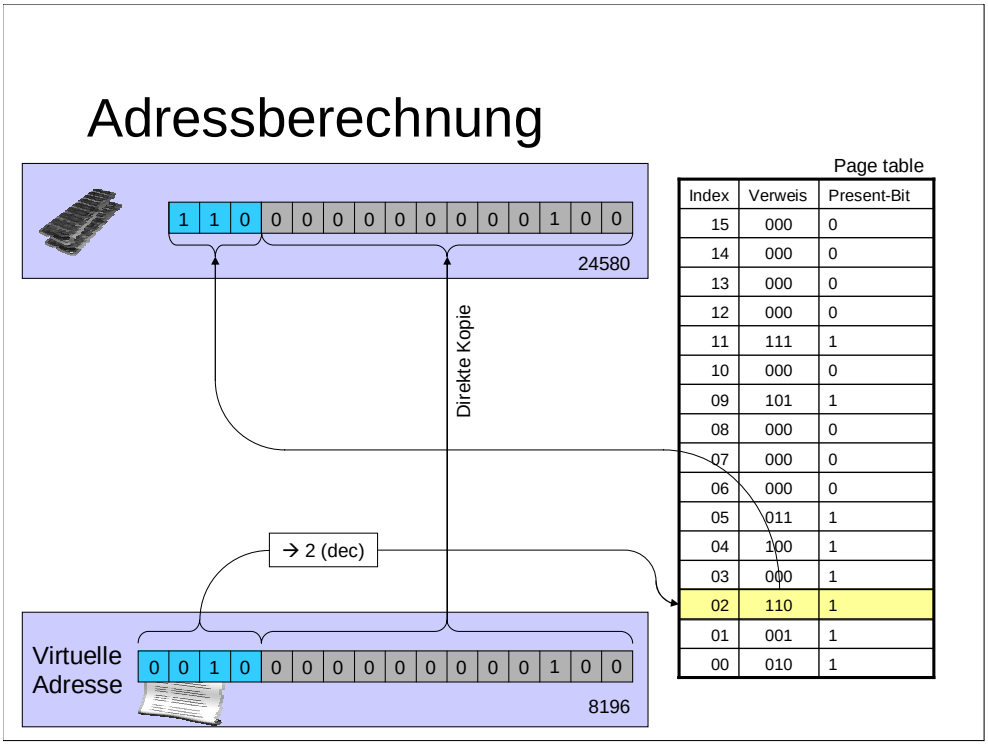
Prinzipiell wird beim Paging zwischen *logischem* Speicher (virtueller Speicher) und *physikalischem* Speicher (Hardware-Speicher) unterschieden. Der logische Speicher definiert den Hauptspeicher aus Programmsicht. Allerdings ist der logische Speicher virtuell und muss deshalb auf den physikalischen Speicher abgebildet werden. Der physikalische Speicher beschreibt den tatsächlich vorhandenen Hauptsspeicher sowie zusätzlich am Hintergrundspeicher ausgelagerten Speicher (z.B. Auslagerungsdatei auf der Festplatte).

Der logische Speicher wird beim Paging in gleich große Stücke aufgeteilt, die so genannten Seiten oder *Pages*. Der physikalische Speicher wird genau gleich aufgeteilt, hier werden die einzelnen Stücke *Frames* genannt. Durch die gleiche Aufteilung der Speicher ergibt sich die Eigenschaft, dass eine Seite und ein Frame genau gleich groß sind und insofern zueinander zugeordnet werden können. Um eine solche Zuordnung durchführen zu können wird eine *Seitentabelle* verwendet. Da jeder Prozess über einen logischen Speicher und insofern über seine Pages verfügt, existiert für jeden Prozess eine derartige Seitentabelle.

! Da in der Seitentabelle die entsprechenden Frames referenziert werden, kann nun keine Fragmentierung mehr entstehen, da es aus Sicht des Prozesses egal ist, ob der verwendete physikalische Speicher zusammenhängt oder weit von einander entfernt liegt. Wichtig ist jedoch, dass die vom Prozess verwendeten *Pages* linear aufsteigend angesprochen werden können.

ⓘ Die Zugriffszeiten auf die physischen Speicherzellen und somit auf die einzelnen Frames sind immer identisch. Insofern entstehen keine Einbußen in Hinblick auf die Effizienz, wenn die einzelnen Frames nicht nebeneinander liegen. Allerdings ist der Zugriff auf die Seitentabelle eine zeitkritische Aufgabe und muss entsprechend optimiert werden. Moderne Prozessoren verwenden zu diesem Zweck spezielle Hardware-Register und Cache-Techniken, die einen effizienten Zugriff auf die Seitentabelle erlauben.

Adressberechnung



Die eigentliche Umsetzung von virtuellen in reale Adressen mittels einer Seitentabelle soll anhand eines Beispiels erläutert werden.

In unserem Beispiel gehen wir davon aus, dass ein Programm die Adresse ‚8196' ansprechen möchte (‚0010000000000100' binär). Die Übersetzung der Adresse in eine reale Adresse sollte effizient passieren, wird jedoch über die Suche in einer Seitentabelle durchgeführt. Deswegen ist es wichtig, die Seitentabelle so klein als möglich zu halten. Deshalb wird in der Realität oftmals die virtuelle Adresse in Unterbereiche aufgeteilt, die zum einen den Index in der Seitentabelle spezifizieren und zum anderen einen Offset im Speicher kennzeichnen. In unserem Beispiel teilen wir die 16-Bit lange virtuelle Adresse in eine 4-Bit lange Seitennummer und einen 12-Bit langen Offset.

Die 4-Bit lange Seitennummer dient als Index in der Seitentabelle, wodurch wir in der Seitentabelle 16 Seiten ansprechen können ($2^4 = 16$). Der Index kennzeichnet den Eintrag in der Seitetabelle, über den wiederum die Adresse des entsprechenden Frames im Hauptspeicher herausgefunden werden kann. Zusätzlich zeigt das *Present*-Bit an, ob der gewünschte Frame im Hauptspeicher verfügbar ist. Die restlichen 12 Bit kennzeichnen den Offset (Position) innerhalb einer Seite. Insofern ist es logisch, dass die in unserem Beispiel gewählten Seiten eine Größe von jeweils 12 Bit haben, also 4 KB (2^{12} = 4096 Byte). Der zugehörige Frame wird wiederum zusammengebaut aus dem Verweis in der Seitentabelle und dem Offset der virtuellen Adresse. Mittels dieser Technik können wir nun effizient den gesamten zur Verfügung stehenden Speicher adressieren.

Effizienz-Problematik

Problematik

- Pro Prozess eine Seitentabelle
  - Hoher Speicherbedarf
  - Page table wird selbst im Speicher gehalten
- Effizienz
  - Doppelter Zugriff → Speicherzugriffe dauern doppelt so lange wie direkter Zugriff
- Lösung
  - Cache Memory in der MMU: Translation Lookaside Buffer (TLB)
  - Speicherung kleiner Anzahl von Einträgen in TLB

Aus dem vorigen Beispiel ist leicht ersichtlich, dass Paging mit einigen Effizienzproblemen zu kämpfen hat. Zum einen ist es notwendig, pro Prozess eine eigene Seitentabelle zu führen, die wiederum im Speicher abgelegt werden muss. Im vorigen Beispiel wurde der Einfachheit halber davon ausgegangen, dass das System über 16-Bit Adressen verfügt. Reale Systeme verfügen allerdings über 32-Bit oder sogar 64-Bit Adressen wodurch sich der Speicherbedarf stark erhöht. Bei längeren Adressen werden entweder mehrere Bits als Index für die Seitentabelle verwendet (wobei bei jedem weiteren Bit die Seitentabelle aufs doppelte anwächst) oder aber größere Page- und Framegrößen verwendet werden, was oftmals sehr ineffizient ist da diese nicht vollständig ausgenutzt werden und somit Speicher verschwenden.

Darüber hinaus werden die Seitentabellen selbst im Speicher gehalten. Gerade dieser Umstand führt dazu, dass im Vergleich zum direkten Speicherzugriff der Zugriff über Paging doppelt so lange dauert, da zweimal auf den Speicher zugegriffen werden muss (einmal auf die Seitentabelle und ein zweites mal auf den angeforderten Speicher).

Eine Verbesserung des Zugriffs kann durch die Verwendung eines speziellen Cache-Speichers in der MMU erzielt werden. Da Programme meist einige Speicheradressen öfters nutzen als andere (z.B. die der Main-Routine), werden eine Reihe von Adressen in einem Cache in der MMU abgelegt und müssen so nicht bei jedem Zugriff erneut berechnet werden. Insofern fällt bei diesen Adressen der Zugriff auf die Seitentabelle weg, da die MMU die im Cache abgelegten Adressen sofort auflösen kann.

Translation Lookaside Buffer

Translation Lookaside Buffer

- Sehr schneller Speicher der eine bestimmte Anzahl (ca. 64) Referenzeinträge zwischenspeichert
  - Information zu einer Speicherseite
    - Virtual Page Nr.
    - Modified Bit
    - Permission (read/write/execute)
    - Physical Page Frame
    - Valid Bit → Eintrag gehört zum aktuellen Prozess
- Beim Aufsuchen einer Adresse wird zuerst der TLB (schneller Speicher) auf Einträge durchsucht
- Falls kein Eintrag vorhanden, erfolgt Suche über herkömmliche Page Table (langsamer Speicher)

Wie zuvor erwähnt ist die mehrstufige Berechung einer physikalischen Speicheradresse aus einer virtuellen Adresse eine zeitintensive Aufgabe, die die Effizienz stark beeinträchtigen kann. Um die Effizienz zu steigern, werden die berechneten Adressen gepuffert und können so bei einer erneuten Anfrage ohne Neuberechnung sofort aufgelöst werden. Die Pufferung erfolgt im so genannten *Translation Lookaside Buffer* (TLB), der Teil der MMU ist. Der TLB kann eine begrenzte Menge (üblicherweise 64 Einträge) aufnehmen und dadurch die Ausführung von Speicherzugriffen erheblich beschleunigen.

Der TLB kann als eine Tabelle mit Informationen zu Speicherseiten gesehen werden, wo folgende Informationen abgelegt sind:

- Virtual Page Nr.: Die virtuelle Seitennummer, die aufgelöst werden soll.
- Modified Bit: Kennzeichnet ob die Seite verändert wurde und insofern erneut berechnet werden muss.
- Permission: Gibt an, wie auf die Speicherseite zugegriffen werden kann.
- Physical Page Frame: Die zugeordnete physikalische Framenummer.
- Valid Bit: Gibt an ob die virtuelle Seite zum aktuellen Prozess gehört (und insofern der Eintrag benutzt werden darf) oder nicht.

Wird nun eine Adresse über den Prozess angefordert, so wird zuerst der TLB auf einen passenden Eintrag durchsucht. Falls kein Eintrag gefunden wird, wird die angeforderte Seite auf herkömmliche Weise aufgelöst und dann im TLB abgelegt.

Mehrstufige Seitentabellen

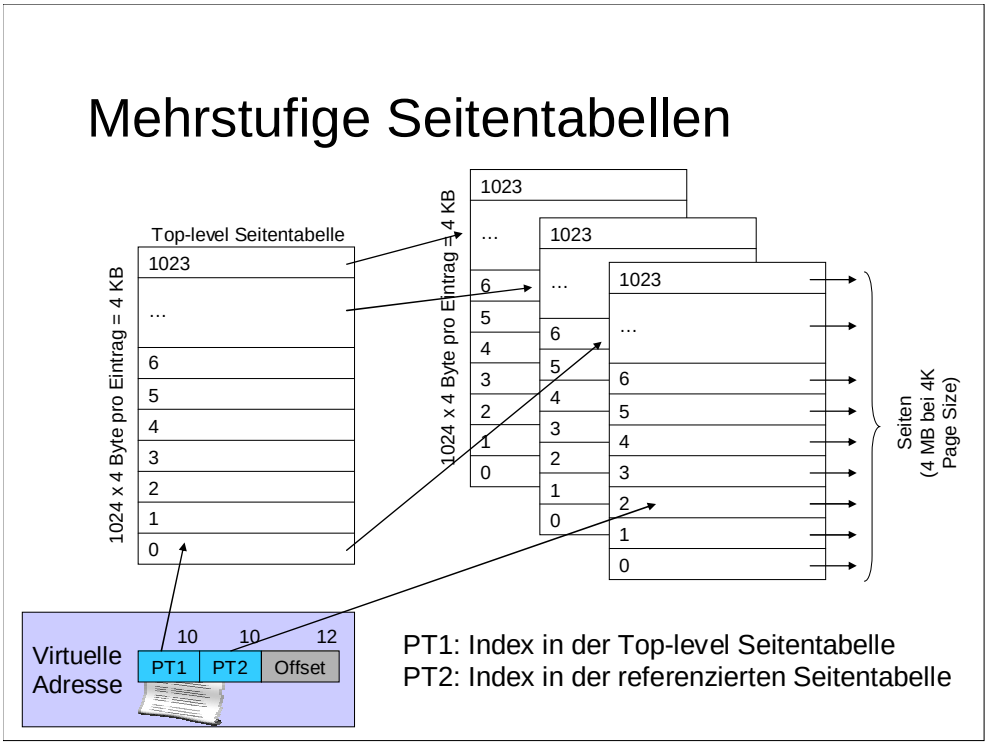
Mehrstufige Seitentabellen

- Seitentabellen werden bei 32-bit Adressraum sehr groß
  - 4K Seitengröße
  - Seitennummer ist 20 Bit lang
  - → 4 MB nur für Seitentabelle
- Ansatz: Nur benötigte Seitentabellen im Speicher halten
  - Mehrstufige Hierarchie
  - Seitentabellen über Indirektion aufgelöst

Die vorher vorgestellten einstufigen Seitentabellen haben das Problem, dass sie sehr schnell groß werden können, vor allem wenn die Seitengröße aus Gründen der effizienten Speicherauslastung gering bleiben soll. In dem vorher erwähnten Beispiel waren die Adressen lediglich 16 Bit lang, wobei eine Seitengröße von 4K im System vorgesehen war. Eine Seitengröße von 4K in einem System, dass 32 Bit für die Adresslänge vorsieht, werden bereits 20 Bit für den Index in der Seitentabelle herangezogen woraus sich eine 4 MB große Seitentabelle ergibt.

Um zu vermeiden, dass im System immer sehr große Seitentabellen gehalten werden, verwenden die meisten Computer heutzutage eine mehrstufige Seitentabelle. Bei diesem Ansatz versucht man, nur die benötigten Seitentabellen im Speicher zu halten und nicht benötigte auszulagern. Dabei wird eine mehrstufige Hierarchie aufgebaut, wobei jeweils nur die benötigten Knoten im Speicher gehalten werden. Die Adressen der Frames und die benötigten Seitentabellen werden dann über Indirektionen in dieser Hierarchie aufgelöst.

Die Funktionsweise solcher Seitentabellen soll nun in weiterer Folge anhand eines Beispiels veranschaulicht werden.



Bei mehrstufigen Seitentabellen wird die virtuelle Adresse in mehrere Bereiche aufgeteilt. In dem hier gezeigten Beispiel wird davon ausgegangen, dass eine zweistufige Hierarchie verwendet wird. Deshalb wird die Adresse in drei Bereiche aufgeteilt:

- PT1: Der Index in der Seitentabelle der ersten Hierarchiestufe (Top-level)
- PT2: Der Index in der Seitentabelle der zweiten Hierarchiestufe
- Offset: Verweis auf den referenzierten Bereich im Frame

Wie in der Grafik ersichtlich, wird nun eine Seitentabelle auf oberster Ebene (Top-level) gehalten, die allerdings nicht auf Frames zeigt, sondern jeweils wiederum auf einen Speicherbereich der eine weitere Seitentabelle hält. Der Index in dieser referenzierten Seitentabelle wird wiederum aus dem zweiten Teil der virtuellen Adresse (PT2) gewonnen. Erst hier steht dann die Nummer des referenzierten Frames, der auf die Adressleitung gelegt werden muss. Ähnlich wie bei der einstufigen Seitentabelle wird innerhalb des Frames die Position mittels des Offsets bestimmt.

Berechnung

In diesem Beispiel verwenden wir nun 32 Bit breite Adressen. Nach wie vor ist die Seitengröße 4 K lang, da lediglich 12 Bit für den Offset verwendet werden (2^{12}). Aus den jeweils 10 Bit für die Seitentabellen können 1024 Einträge referenziert werden (2^{10}). Da jede Seitentabelle nun 1024 Einträge referenziert, die dann in der zweiten Stufe auf eine jeweils 4 K große Seite zeigen, können pro Seitentabelle in der zweiten Hierarchiestufe 4 MB Speicher referenziert werden. Allerdings enthält

die erste Hierarchiestufe 1024 Einträge die jeweils auf solche Seitentabellen der zweiten Hierarchiestufe zeigen, also können insgesamt 1024^2 * 4 K = 4 GB adressiert werden.

Vorteil

Der Vorteil dieser Lösung liegt darin, dass von den vielen möglichen Seitentabellen nicht immer alle im Hauptspeicher liegen müssen. Tatsächlich muss immer die Seitentabelle der ersten Hierarchiestufe im Speicher liegen (Top-Level Seitentabelle), die dann die Möglichkeit hat, auf 1024 weitere Seitentabellen zu referenzieren. Allerdings werden diese Seitentabellen erst dann in den Speicher geladen, wenn diese benötigt werden. Bei der Verwendung von nur einer weiteren Speicherseite in der zweiten Hierarchiestufe können allerdings bereits 4 MB Speicher angesprochen werden! Prozesse, die nicht mehr als 4 MB Speicher benötigen, werden also nur eine Speicherseite in der zweiten Hierarchiestufe verwenden.

Sobald ein Prozess mehr als 4 MB Speicher benötigt, wird eine zweite Speicherseite auf der zweiten Hierarchiestufe angelegt und im Speicher gehalten. Mit einer solchen weiteren Speicherseite können nun weitere 4 MB Speicher adressiert werden.

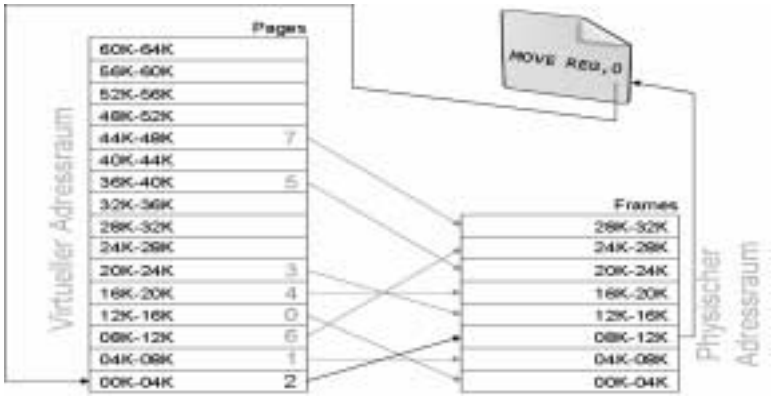
Daraus resultiert, dass diese Lösung sehr flexibel und skalierbar ist, da für Prozesse immer nur so viele Seitentabellen existieren, wie diese für ihre Speicheranforderung benötigen. Kleine Prozesse werden weniger Seitentabellen haben und insofern den Speicher weniger belasten, speicherintensive Prozesse haben allerdings die Möglichkeit bis zu max. 4 GB Speicher zu adressieren (z.B. Datenbank-Prozesse).

Der hier vorgestellte Ansatz ist außerdem beliebig erweiterbar und muss nicht zwangsweise nur aus zwei Hierarchiestufen bestehen. Es ist leicht vorstellbar, eine dritte Hierarchiestufe einzuführen. Allerdings vermindert sich damit der Offset und damit die Seitengröße. Zudem verursacht jede Hierarchiestufe einen Overhead, da ja Verwaltungsarbeit zum Anlegen, Laden und Warten der Hierarchiestufen nötig ist. Deswegen findet man heute bei den meisten 32-Bit Systemen lediglich eine zweistufige Hierarchie.

Fallbeispiel: Paging mit 16 Bit

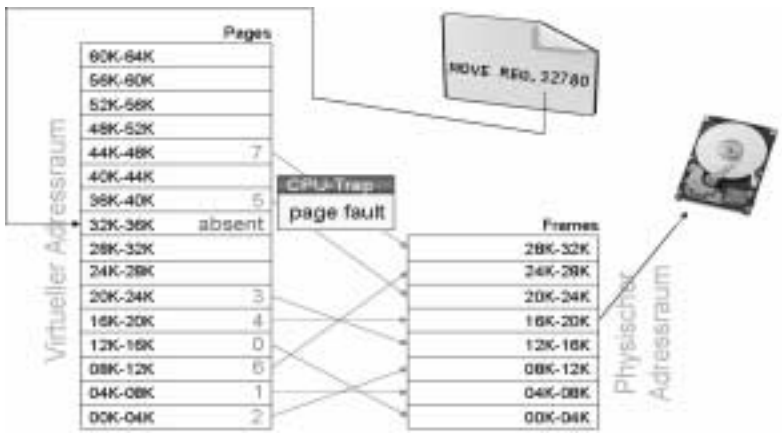
Die Funktionsweise von Paging soll nun nochmals anhand eines Fallbeispiels durchbesprochen werden. Dabei wird der Einfachheit halber davon ausgegangen, dass die Adressbreite aus lediglich 16 Bit besteht und dass nur eine einstufige Seitentabelle verwendet wird.

Im ersten Schritt versucht ein Programm über die Anweisung „MOVE REG,0“ die Seite (*Page*) „0“ zu lesen. Da die Seite 4 K groß ist, wird in der Seitentabelle nachgeschlagen, zu welchem physikalischen Adressraum diese Seite zugeordnet ist.



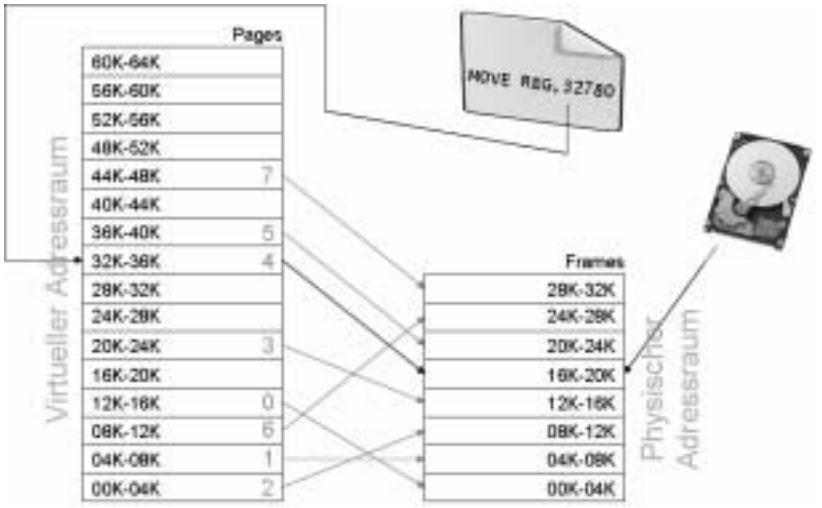
Anhand der Seitentabelle stellt die MMU fest, dass der virtuelle Adressraum „00K-04K“ mittels des Index „2“ auf den physikalischen Frame „08K-12K“ abgebildet wurde. Die MMU fordert diesen Speicherbereich an und stellt die darin abgelegten Daten dem Prozess zu.

Wie zuvor erwähnt, muss die angeforderte Seite nicht zwangsweise bereits auf einen Frame abgebildet sein. Gehen wir davon aus, dass unser Programm im nächsten Statement auf die Adresse „32780“ zeigt, welche in die Seite „32K-36K“ fällt. Dieser Seite ist in der Seitentabelle noch kein Index und somit kein Frame zugewiesen. In diesem Fall unterbricht die MMU die Abarbeitung des Programms indem ein so genannter *Page fault* (Seitenfehler) ausgelöst wird. Dieses Ereignis bedeutet, dass eine angeforderte virtuelle Adresse sich noch nicht im Arbeitsspeicher befindet und insofern zuerst geladen werden muss.



In dem hier gezeigten Beispiel stehen wir allerdings vor dem Problem, dass alle Frames bereits belegt sind und kein Frame zur Verfügung steht.

In diesem Fall muss ein bestehender Frame ersetzt werden, indem er auf den Hintergrundspeicher ausgelagert wird.



In unserem Beispiel wird der Frame „16K – 20K“ auf die Platte ausgelagert und durch den in der Seite „32K-36K“ angeforderten Speicherbereich ersetzt (bestehender Speicher wird von der Festplatte eingelagert).

Da bei den Aus- und Einlagerungen jeweils ein sehr langsamer Speicher (die Festplatte) benutzt wird, ist diese Operation sehr laufzeitintensiv und dadurch teuer. Deshalb stellt sich die Frage, welcher Frame ersetzt werden soll. Um eine möglichst sinnvolle und effiziente Ersetzung zu finden, gibt es unterschiedliche Strategien die nun in weiterer Folge behandelt werden.

Ersetzungsstrategien

Ersetzungsstrategien

- Anforderungen
  - Anzahl der Page Faults sollte möglichst gering sein
  - Effizienter Berechnungsalgorithmus (Hardware)
- Algorithmen
  - Optimal Page Replacement Algorithm
  - First In First Out
  - Least Recently Used
  - Least Frequently Used

Wie zuvor anhand eines Fallbeispiels gezeigt, ist es oftmals notwendig bestehende Frames im Hauptspeicher durch ausgelagerte Frames am Hintergrundspeicher zu ersetzen. Da diese Ersetzung eine sehr teure Operation ist, ist es notwendig eine effiziente Ersetzungsstrategie einzusetzen.

Bei einer effizienten Ersetzungsstrategie sollte die Anzahl der Seitenfehler (*page faults*) so gering als möglich sein. Das bedeutet, dass eine Ersetzung bei einem Seitenfehler nicht unmittelbar später wieder in einem Seitenfehler resultiert, da ein häufig benötigter Frame ersetzt wurde.

Zudem sollte der Algorithmus der den zu ersetzenden Frame bestimmt sehr effizient sein und durch die Hardware berechenbar sein (einfache Rechenoperationen). Eine lange und komplexe Berechnung der zu ersetzenden Seite würde weiters die Laufzeit beeinträchtigen.

In der Theorie gibt es eine Reihe von Ersetzungsalgorithmen, die nun in weiterer Folge vorgestellt werden sollen.

Optimal Page Replacement Algorithm

Optimale Seitenersetzung

- Seiten, die erst später im Code referenziert werden weichen Seiten, die früher referenziert werden.
- Algorithmus muss komplettes Programm kennen um zu wissen, welche Seiten später und welche früher benötigt werden
  - Working Set: Menge aller Seiten die von einem Prozess benötigt werden
- → Unrealisierbar.

Der *Optimal Page Replacement Algorithm* (Optimale Seitenersetzung) versucht Seiten, die erst später im Programm benötigt werden durch Seiten die früher verwendet werden zu ersetzen.

Der Vorteil des Algorithmus liegt darin, dass während der Programmausführung immer die Seiten und Frames zur Verfügung stehen, die gerade benötigt werden.

Allerdings muss der Algorithmus das komplette Programm kennen um zu wissen wann welche Seiten verwendet werden und insofern auch welche Frames wann benötigt werden. Der minimale Anteil an Seiten, die sich im Speicher befinden muss um für einen Prozess eine solche Entscheidungen zu treffen wird als *Working Set* bezeichnet. In diesem Fall würde das Working Set jedoch aus allen Seiten des Prozesses bestehen, damit die optimale Strategie abgeleitet werden kann.

! Leider ist dieser Algorithmus unmöglich zu realisieren, da davon ausgegangen wird, dass das Betriebssystem in die Zukunft blicken kann und deshalb weiß, wie sich der Prozess verhalten wird. Da aber größtenteils die Arbeitsweise des Prozesses vom Benutzer abhängt, ist es für das Betriebssystem unmöglich zu wissen wann welche Speicherbereiche benötigt und verwendet werden.

First In – First Out (FIFO)

FIFO

- Seiten, die sich am längsten im Speicher befinden werden ersetzt
- Nachteil: Berücksichtigt nicht wie oft auf eine Seite zugegriffen wurde
- Erfahrung: Gewisse Speicherbereiche werden sehr oft benötigt (Main Event Loop)

Diese Strategie verfolgt den Ansatz, dass die Seiten ersetzt werden, die sich bereits am längsten im Speicher befinden. Dabei wird davon ausgegangen, dass alte Speicherbereiche nicht mehr in Verwendung sind und deshalb leicht ersetzt werden können.

Der Vorteil dieser Lösung liegt darin, dass diese sehr einfach und effizient zu implementieren ist, da einfach pro Eintrag mitgespeichert werden muss, wann dieser erzeugt wurde und dann der Eintrag mit dem ältesten Datum entfernt wird.

Der Nachteil dieses Ansatzes ist jedoch, dass nicht berücksichtigt wird wie oft auf eine Seite zugegriffen wurde. So zeigt die Erfahrung, dass gewisse Speicherbereiche eines Programms öfters benötigt werden als andere. Zum Beispiel wird die *Main Event Loop* (die Routine, die Benutzereingaben zustellt bzw. die Steuerung des Programms vornimmt) als erstes erzeugt und während der gesamten Lebensdauer des Programms verwendet. Eine solche Routine würde öfters ersetzt werden und immer wieder erneut geladen werden und insofern sehr viele unnötige Seitenersetzungsfehler produzieren.

FIFO – Second Chance

FIFO - Second Chance

- Erweiterung der FIFO Strategie
- Pro Seite wird ein used-bit mitgeführt, dass bei einem Seitenzugriff gesetzt wird
- Soll eine Seite nach FIFO Strategie ersetzt werden, wird zunächst geprüft ob das used-bit gesetzt ist
  - Ja: used-bit wird zurückgesetzt, Seite kommt an das Ende der Liste, der Algorithmus überprüft die nächste Seite
  - Nein: Ersetzung erfolgt

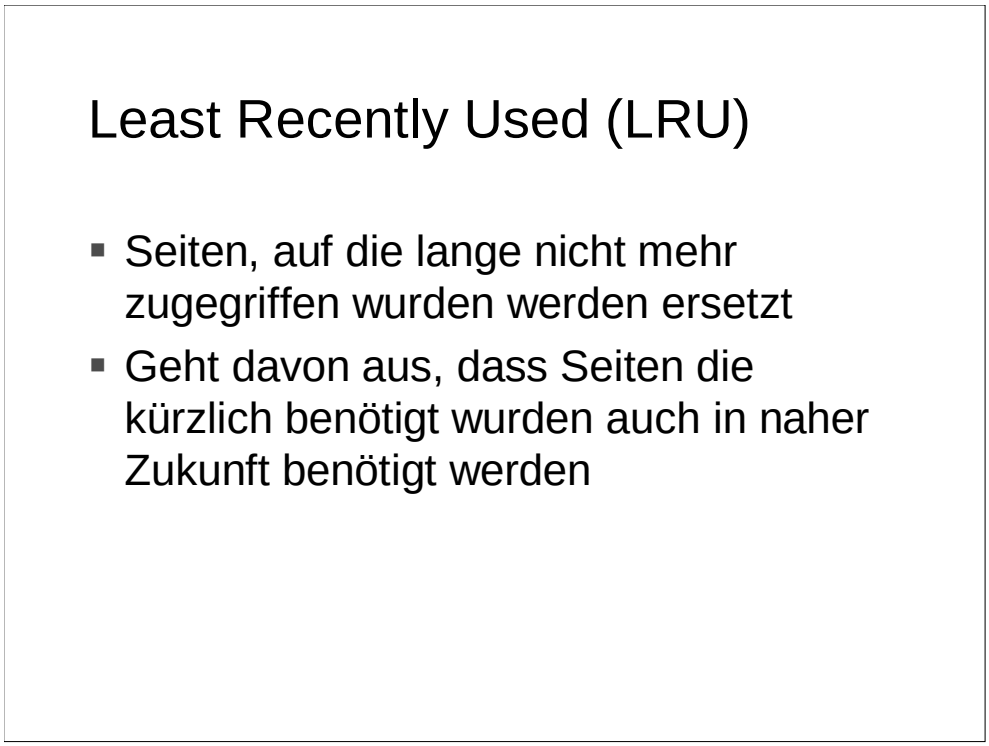
Dieser Ansatz ist eine Weiterentwicklung des reinen FIFO-Algorithmus und berücksichtigt indirekt den letzten Seitenzugriff.

Pro Seite wird ein Bit mitgeführt (*used-bit*), das die Verwendung der Seite kennzeichnet. Sobald eine Seite vom Programm aus referenziert wird, wird dieses Bit gesetzt und insofern gekennzeichnet, dass die Seite in Verwendung ist. Nach wie vor kommt bei einem Seitenfehler die FIFO-Strategie zum Einsatz, d.h. die älteste Seite ist ein Ersetzungskandidat. Anders als bei der reinen FIFO-Strategie wird allerdings vorher noch überprüft, ob das used-bit gesetzt ist. Falls dieses Bit gesetzt wird, wird das Bit zurückgesetzt (unused), die Seite allerdings nicht ersetzt sondern mit einem aktuellen Zeiteintrag versehen und insofern an das Ende der Liste gesetzt – der Eintrag erhält eine zweite Chance. Der Algorithmus sucht dann nach dem FIFO-Prinzip den nächsten Ersetzungskandidaten. Ist bei einem Ersetzungskandidaten das used-bit allerdings nicht gesetzt, so erfolgt die Ersetzung.

Der Vorteil dieses Ansatzes ist, dass häufig verwendete Seiten nicht sofort ersetzt werden sondern immer eine zweite Chance erhalten, da ja ihr used-bit gesetzt ist. So werden häufig benutzte Seiten nie verdrängt.

Der Nachteil des Ansatzes liegt darin, dass nicht die Referenzhäufigkeit (wie viele Zugriffe von unterschiedlichen Prozessen oder Routinen zeigen auf diesen Eintrag) berücksichtigt wird. Einträge, die von vielen Prozessen oder Routinen referenziert werden sollen schnell verfügbar sein, da sie meist zentrale Routinen darstellen.

Least Recently Used (LRU)



Die *Least Recently Used* (LRU) Strategie ersetzt die Seiten, auf die lange nicht mehr zugegriffen wurden. Dabei geht diese Strategie davon aus, dass Seiten die kürzlich benötigt wurden auch in naher Zukunft benötigt werden und ersetzt Seiten, die schon länger nicht mehr benutzt wurden.

Der Vorteil dieser Strategie liegt in der Einfachheit der Implementierung, da lediglich zusätzlich ein Zähler pro Seite mitgeführt werden muss, der darüber Aufschluss gibt, wie lange eine Seite nicht mehr referenziert wurde. Zusätzlich berücksichtigt diese Strategie auch das Alter des Eintrags.

Einer der wesentlichsten Nachteile dieses Ansatzes ist, dass genauso wie beim FIFO-Second Chance nicht die Referenzhäufigkeit berücksichtigt wird. Zudem geht der Algorithmus auch rein davon aus, dass kürzlich zugegriffene Seiten auch in Zukunft oft benutzt werden. Zentrale Routinen, die in aufwändige Unterprogramme wechseln, könnten somit ersetzt werden.

Implementierung des LRU Algorithmus

LRU - Implementierung

- Mitführen eines Zählers pro Seite
  - Seite hat ein used-bit und einen Zähler
  - Bei Referenzierung wird used-bit gesetzt
  - In regelmäßigen Intervallen wird der Zähler aller Seiten mit gesetztem used-bit auf 0 gesetzt
  - Bei allen anderen Seiten wird der Zähler um eins erhöht
  - Bei Page Fault wird Seite mit höchstem Zählerwert ersetzt

Der LRU-Algorithmus kann auf unterschiedliche Weise implementiert werden. Allerdings ist das Protokollieren der Zugriffseiten auf eine Seite ein aufwändiger Prozess. Insofern hat sich die Variante des Mitführens eines Zählers pro Seite bewährt.

Bei dieser Variante wird jeder Seite ein *used-bit* und ein Zähler zugeordnet. Wie beim Second Chance Algorithmus wird bei jeder Referenzierung das *used-bit* gesetzt. In regelmäßigen Intervallen wird der Zähler aller Seiten mit gesetztem *used-bit* auf 0 zurückgesetzt und das *used-bit* gelöscht. Die Zahl ‚0' kennzeichnet dabei, dass die Seite in jüngster Zeit benutzt wurde. Bei allen anderen Seiten, dessen *used-bit* nicht gesetzt ist, wird der Zähler um eine Werteinheit erhöht (Seite wurde in diesem Zyklus nicht benutzt).

Sobald ein Seitenfehler auftritt und deshalb eine Ersetzung stattfinden muss, wird die Seite mit dem höchsten Zählerwert (Seite die am Längsten in keinem Zyklus mehr benutzt wurde) ersetzt. Der Zählerwert gibt insofern an, wie lange eine Seite nicht mehr benutzt wurde.

Least Frequently Used

Least Frequently Used (LFU)

- Seiten, auf die am wenigsten benutzt wurden werden ersetzt
- Jeder Seite wird ein Zähler zugeordnet, bei Zugriff wird Zähler erhöht
- Bei Page Fault wird Seite mit niedrigstem Zugriffswert ersetzt
- Besseres Ergebnis wenn zudem Zugriffswert aller Seiten in regelmäßigen Abständen auf 0 gesetzt wird
  - Seiten, die früher oft benutzt jedoch länger nicht mehr benötigt wurden bleiben sonst im Speicher

Die bis dato gezeigten Strategien (FIFO, FIFO-Second Chance, LRU) sind davon ausgegangen, dass die Seiten ersetzt werden, die am ältesten sind bzw. am längsten nicht mehr benutzt wurden. Allerdings wurde nicht die Häufigkeit der Benutzung der einzelnen Seiten berücksichtigt. Tatsächlich verfügen aber viele Programme über Routinen, die zwar nur hin und wieder aufgerufen werden, dafür aber sehr häufig (z.B. Menüsteuerung).

Der Least Frequently Used Algorithmus versucht, die Seiten zu ersetzen, die am wenigsten benutzt wurden. Dazu führt er pro Seite einen Zähler mit, der die Anzahl der Zugriffe speichert. Bei jedem Zugriff wird der Zähler erhöht. Bei einem Seitenzugriffsfehler wird die Seite mit dem niedrigsten Zählerwert (die Seite auf die am wenigsten oft zugegriffen wurde) ersetzt.

Das Problem bei diesem Algorithmus liegt darin, dass Seiten die zu Beginn oft benutzt werden (z.B. beim Laden) sich sehr lange erhalten obwohl diese nicht mehr benötigt werden und insofern wertvollen Platz in der Seitentabelle blockieren. Eine Optimierung könnte dadurch erzielt werden, dass die Zugriffswerte aller Seiten in regelmäßigen Abständen auf 0 zurückgesetzt werden oder dass in regelmäßigen Intervallen die Zugriffswerte wieder reduziert werden.

Optimierung der Algorithmen

Optimierungen der Algorithmen

- Dirty Bit
  - Kennzeichnet ob Seite bei Zugriff modifiziert wurde
  - Wenn ja, muss die Änderung auf den Auslagerungsspeicher repliziert werden → langsamer Speicher, teure Operation!
  - Wenn nein, braucht diese Änderung nicht erfolgen

Wie anhand der verschiedenen Algorithmen gezeigt wurde, haben alle Strategien ihre Vorteile und Nachteile. Der optimale Algorithmus kann aufgrund der Notwendigkeit der Kenntnis des kompletten Programmablaufs nicht realisiert werden. Zusätzlich berücksichtigen die Algorithmen nicht, dass etwaige Änderungen beim Zugriff auf eine Seite auf den Auslagerungsspeicher repliziert werden müssen was wiederum eine sehr teure Operation ist.

Deshalb können die Algorithmen durch die Einführung eines *Dirty Bits* zusätzlich optimiert werden. Dieses Bit kennzeichnet, ob eine Seite bei einem Zugriff modifiziert wurde und deshalb die Änderung auf den Hintergrundspeicher repliziert werden muss. Diese Replizierung hat bei einer Ersetzung und Auslagerung zu erfolgen.

Die Idee bei diesem Ansatz geht davon aus, dass sich Speicherbereiche, die sich bereits im Auslagerungsspeicher befinden und für die Verwendung eingelagert wurden, allerdings nicht verändert wurden, nicht erneut ausgelagert werden müssen da der Auslagerungsspeicher bereits den exakt gleichen Eintrag enthält. Insofern kann die Aktualisierung des langsamen Auslagerungsspeichers wegfallen.

Wird jedoch ein Speicherbereich geändert, muss dieser bei der Verdrängung auch den zugehörigen Auslagerungsspeicher aktualisieren. Änderungen werden durch das Setzen des *Dirty Bits* gekennzeichnet. Ist dieses Bit gesetzt so erfolgt die Aktualisierung des Auslagerungsspeichers, ansonsten kann diese teure Operation wegfallen.

Paging vs. Swapping

Paging vs. Swapping

- Paging
  - Ein-/Auslagerung von Speicherseiten
- Swapping
  - Ein-/Auslagerung vollständige Prozesse
  - Datensegment des alten Prozesses wird bei Verdrängung vollständig auf Hintergrundspeicher ausgelagert, Datensegment des neuen Prozesses wird von Hintergrundspeicher eingelagert
  - Programme, die größer als Hauptspeicher sind, können nicht ausgeführt werden

Wie in diesem Kapitel gezeigt, liegt der große Vorteil von Paging in der Möglichkeit, Programmen größere virtuelle Speicherbereiche zur Verfügung zu stellen als real im Computer vorhanden. Dies geschieht durch Aufteilen der Speicherbereiche in gleich große Seiten und Frames, die jeweils durch Zuhilfenahme von Hintergrundspeicher ein- und ausgelagert (ersetzt) werden können. Der Vorteil von Paging besteht darin, dass es durch die Verwendung von Seitentabellen nie zur Fragmentierung des Hauptspeichers kommt. Allerdings hat Paging einige Effizienzprobleme und aufgrund der Tatsache, dass erst bei Seitenfehler eine Ersetzung stattfindet ein nicht deterministisches Laufzeitverhalten. Gerade diese Tatsache macht Paging für den Einsatz in Echtzeitbetriebssystemen unbrauchbar.

Dem gegenüber gibt es auch den Ansatz von *Swapping*. Hier werden komplette Speichersegmente zwischen Arbeitsspeicher und Hintergrundspeicher aus- und eingelagert. Allerdings wird bei diesem Ansatz immer das komplette zum Prozess gehörige Speichersegment aus- und eingelagert. Bei einer Verdrängung eines alten Prozesses wird das Datensegment vollständig auf den Hintergrundspeicher ausgelagert und das Datensegment des neuen Prozesses vom Hintergrundspeicher eingelagert. Das bedeutet, dass beim Swapping die Daten eines Prozesses entweder vollständig ausgelagert oder vollständig im Hauptspeicher enthalten sind. Diese Eigenschaft unterscheidet das Swapping vom Paging bei dem nur einzelne Speicherseiten ein- und ausgelagert werden. Die Konsequenz die sich aus diesem Verhalten ergibt ist diejenige, dass bei der Verwendung von Swapping keine Programme ausgeführt werden können, die mehr Arbeitsspeicher benötigen würden als real vorhanden ist. Zudem erhöht sich der Zeitbedarf bei einem Kontext-Switch, da nun ja die Datensegmente zusätzlich ein- und ausgelagert werden müssen.