Grundlagen und Geschichte

Definition Betriebssystem (Tanenbaum)

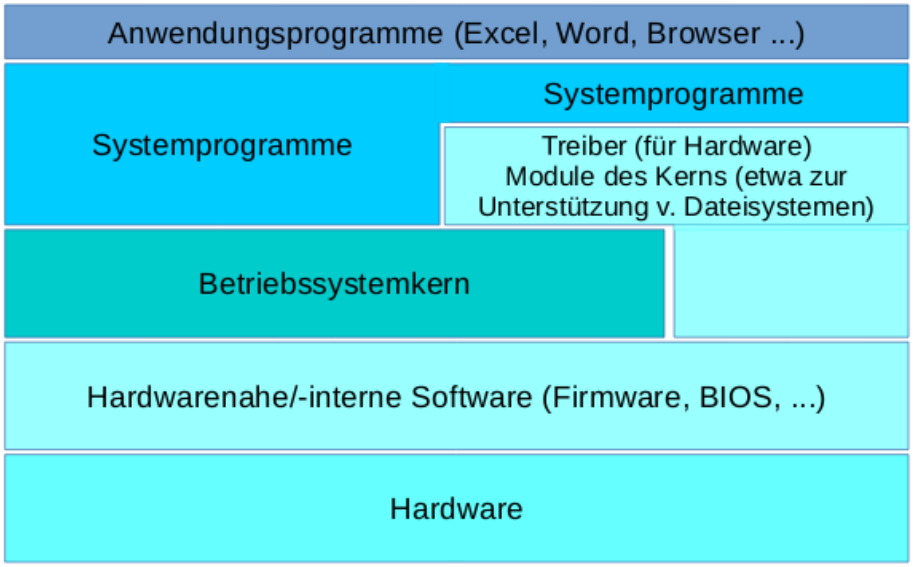
Ein Betriebssystem ist eine Software, die die Aufgabe hat, vorhandene Geräte zu verwalten und Benutzerprogrammen eine einfache Schnittstelle zur Hardware zur Verfügung zu stellen.

Betriebssysteme übernehmen zwei an sich unabhängige Funktionen. Einerseits die Erweiterung der Hardware und zum anderen die Verwaltung der Ressourcen.

Aufgaben eines Betriebssystems (Harris)

- Prozessverwaltung: den laufenden Prozessen Rechenzeit zuteilen und Ressourcen zuweisen und mehrere Prozesse gleichzeitig verwalten können
- Speicherverwaltung: Speicher für sich selbst und für laufende Prozesse zuteilen, belegen und wieder freigeben
- Verwaltung des Dateisystems: Organisation der Datenspeicherung und Zugriffsrechte auf die Dateien verwalten
- Verwaltung von Geräten: Interaktion mit angeschlossenen Geräten mithilfe von Treibern ermöglichen und Zugriffe koordinieren und überwachen

Aufbau eines Betriebssystems



Unterschied zwischen Anwendungs- und Systemprogrammen

Anwendungsprogramme sind Programme, die für den Endbenutzer bestimmt sind. Sie haben meistens eine recht simple Oberfläche und haben wenig Zugriffsrechte.

Systemprogramme sind für die Administration und Konfiguration des Systems gedacht. Sie werden von Administratoren verwendet und haben meistens höhere Zugriffsrechte und Prioritäten.

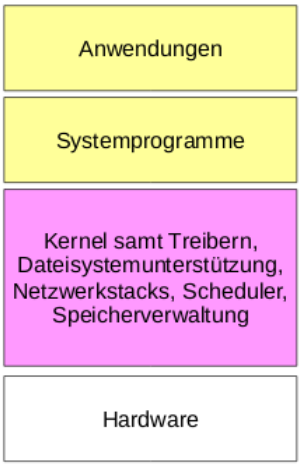
Kern eines Betriebssystems

Der Kern eines Betriebssystems (auch Kernel genannt) greift direkt auf die Hardware zu (Kommunikation mit Firmware und BIOS). Er hat die höchste Priorität und die höchsten Berechtigungen.

Der Kernelspace ist Code des Kernels, der in einem eigenem Speicherbereich mit höheren Privilegien ausgeführt wird.

Der Userspace ist Code, der außerhalb des Kernels ausgeführt wird und Speicherbereiche verwendet, die nicht vom Kernel verwendet werden. Er hat geringere Prioritäten.

Monolithische Kernel

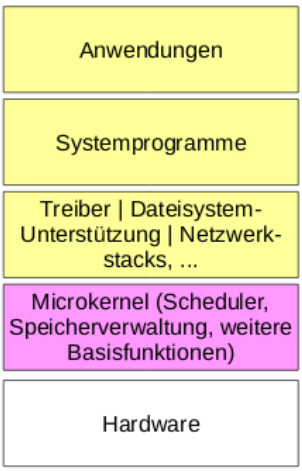


Ein monolithischer Kernel integriert alle Systemfunktionen. Er beinhaltet auch Funktionen, die nicht essentiell für das Betriebssystem sind. Einige davon werden eventuell nicht einmal verwendet.

Vorteil: Es werden keine extra Programme für Treiber benötigt und durch die direkte Integration ergibt sich ein Geschwindigkeitsvorteil.

Nachteil: Eine Änderung am Kernel ist nur durch einen Neustart möglich. Außerdem führt das Ausfallen einer Komponente zum Absturz des gesamten Kernels.

Mikrokernel



Ein Mikrokernel beinhaltet nur die essentiellen Funktionalitäten des Kernels. Alle restlichen Funktionen werden als Services in den Userspace ausgelagert.

Vorteil: Austausch von Komponenten ist während der Laufzeit möglich. Stürzt eine Komponente ab, läuft der restliche Kernel weiter.

Nachteil: Das Auslagern in den Userspace benötigt Rechenzeit.

Hybridkernel (Makrokernel)

Ein Hybridkernel ist eine Kombination aus monolithischem Kernel und Mikrokernel. Es sind relative viele Funktionen durch den Kernel abgedeckt, es können aber auch Funktionen aus dem Userspace nachgeladen werden. Der Hybridkernel bringt dadurch eine bessere Performance und eine höhere Stabilität und hat sich bei modernen Betriebssystem durchgesetzt.

Erste Generation (ca. 1945 – 1955)

- Begriff „Betriebssystem“ noch unbekannt
- einzelne Personengruppen kümmerten sich um alles
- Programmierung durch Maschinensprache bzw. Verdrahtung
- Programmiersprachen noch unbekannt

Zweite Generation (ca. 1955 - 1965)

- Unterscheidung zwischen Entwicklern, Herstellern, Operateuren, Programmierern und Wartungspersonal
- Einsatz hauptsächlich in Großunternehmen und in Universitäten
- Programmierung über Lochkarten, Ausgabe auf Endlospapier
- Stapelverarbeitung ohne Benutzerinteraktion

Dritte Generation (ca. 1965 – 1980)

- Einführung von Multiprogramming sorgte für Reduktion der Wartezeit
- Spooling verkürzte die Ausführzeit durch effizienteres Nutzen der Speichermedien
- Timesharing ermöglicht das scheinbar gleichzeitige Arbeiten mehrerer Benutzer
- erstes Aufkommen von deutlich günstigeren Minicomputern

Vierte Generation (seit ca. 1980)

- Benutzerfreundlichkeit stand als wichtiges Ziel im Mittelpunkt
- Einführung von Maus und grafischen Benutzeroberflächen
- Netzwerkfähigkeit und Ermöglichung von verteilter Programmausführung
- Betriebssystem und Hardware auch für Privathaushalte

Multiprogramming

Bei Multiprogramming wird der Speicher partitioniert, damit mehrere Jobs gleichzeitig geladen werden können. Es kann immer nur ein Job gleichzeitig ausgeführt werden. Ein Wechsel zwischen den Jobs findet immer dann statt, wenn der aktuelle Job auf Ein- oder Ausgabe wartet. Somit wird keine Rechenzeit der CPU für Wartezeit verschwendet und die CPU-Auslastung wird maximiert.

Spooling

Spooling beschreibt den Vorgang, dass Daten von einem langsamen Medium (z.B. Magnetband) auf ein schnelles Medium (z.B. Festplatte) zwischengelagert werden, damit auf diese dort schneller zugegriffen werden kann. Während Job A ausgeführt wird, kann Job B parallel dazu bereits auf das schnellere Medium eingelesen werden, um anschließend schneller ausgeführt werden zu können.

Timesharing

Durch Timesharing wurden die ersten Mehrbenutzersysteme realisiert. Die angemeldeten Benutzer bekommen periodisch abwechselnd Rechenzeit zugeteilt, abhängig davon wie aktiv sie sind. Der Jobwechsel erfolgt so schnell, dass der Eindruck entsteht, dass keine anderen Nutzer am System arbeiten.

Arten von Betriebssystemen

Bezeichnung	Besondere Merkmale	Beispiele für Einsatzgebiete	Beispiele für Betriebssysteme
Mainframe Betriebssysteme	sind für Großrechner ausgelegt und haben hohe Anforderungen an Performance, Datenkapazität und Zuverlässigkeit	Große Webserver B2B-Anwendungen Flugbuchungen Große Banken	IBM OS/360 Unix-Derivate
Server Betriebssysteme	Gleiche Anforderungen wie Mainframe Betriebssysteme aber in geringerer Ausprägung	Webserver Workstations	Linux Unix-Derivate BSD-Derivate Windows Server
PC-Betriebssysteme	Stellen Multiprozessor-Unterstützung bereit und dienen der Bearbeitung von digital unterstützten Aufgaben	Heimcomputer Bürocomputer	Windows Mac OS X Linux
Echtzeit-Betriebssysteme	Können zeitkritische Prozesse verarbeiten	Produktionsanlagen mit Robotern Gebäudeautomation	VxWorks QNX
Betriebssysteme für eingebettete Systeme	Müssen mit beschränkten Ressourcen kontextspezifische Dienste ermöglichen	Microcontroller Smartphones TV-Geräte	Android iOS WatchOS tvOS
Betriebssysteme für Sensorknoten	Sollen wartungsarme Langzeitbetriebe mit knappen Ressourcen ermöglichen	Temperatursensoren Rauchmelder	TinyOS RIOT Contiki
Betriebssystem für Chipkarten	Müssen unter extremer Ressourcenknappheit auf Chipkarten laufen	EC-Karten Kreditkarten Schlüsselkarten	SECCOS STARCOS CardOs
Betriebssysteme für Spielekonsolen	PC-Betriebssystem mit optimierter Grafikperformance	Spielekonsolen	Nintendo Switch OS PlayStation Vita OS

Prozesse und Threads

Definition Prozess und Multitasking

Ein Prozess ist ein Programm in Ausführung. Wenn mehrere Prozesse gleichzeitig laufen, spricht man von Multitasking. Das parallele Ausführen von mehreren Prozessen ist nur dann möglich, wenn auch mindestens zwei Prozessoren oder Prozessorkerne zur Verfügung stehen. Wenn nur ein Prozessor vorhanden ist, kann dennoch Quasiparallelität erreicht werden. Bei Quasiparallelität wechselt die CPU schnell zwischen den Prozessen, dass der Eindruck von paralleler Prozessverarbeitung entsteht.

Befehlszeiger

Der Befehlszeiger ist eine Hardwarekomponente des Computers. Er zeigt auf die aktuell auszuführende Anweisung im Speicher eines Prozesses. Für jeden Prozess A muss dieser Hardware-Befehlszeiger auf die Speicherstelle gesetzt werden, die nach einer Unterbrechung durch einen anderen Prozess B die Weiterausführung des Prozesses A an der korrekten Speicherstelle ermöglicht.

Virtuelle Befehlszeiger stehen für jeden Prozess zur Verfügung. Sie speichern die Position des Hardware-Befehlszeigers für einen Prozess zwischen, während ein anderer Prozess ausgeführt wird.

CPU-Auslastung

Auf einem System laufen n Prozesse und es steht eine CPU zur Verfügung. Ein Prozess wartet einen Teil p seiner Zeit auf Ein- und Ausgabe. Die Wahrscheinlichkeit, dass alle n Prozesse auf Ein- und Ausgabe warten (d.h. die CPU wäre unbeschäftigt) beträgt somit p^n. Damit ergibt sich für die CPU-Auslastung durchschnittlich:

$$Auslastung = 1 - p^n$$

Speedup durch Parallelisierung

Bei parallelisierbarem Code kann man mit zusätzlichen Prozessoren eine Beschleunigung erreichen. Beim Ausführen eines Codes mit einem parallelisierbaren Anteil f kann man durch die Verwendung von p CPU-Kernen den folgenden Speedup S_p erreichen:

$$S_p = \frac{1}{(1-f)+(f \div p)}$$

Der Idealfall $S_p = p$ wird in der Praxis allerdings nicht erreicht, da ein Prozess nie zu 100% parallelisierbar ist und ein Overhead durch die Kommunikation zwischen Teilprozessen entsteht (bspw. durch den Abgleich von Zwischenergebnissen).

Unterschiede von Prozessen

Keine zwei Prozesse sind „gleich“. Zwei Prozesse können zwar gleiche Programme repräsentieren, aber werden ggf. unterschiedliche Daten verwenden. In jedem Fall bekommen sie eine unterschiedliche Prozess-ID zugewiesen und einen eigenen Speicherbereich.

Es gibt verschiedene Arten von Prozessen:

- Vordergrundprozesse: aktives Interface zum Benutzer
- Hintergrundprozesse: es gibt kein direkt zugreifbares Interface
- Systemprozesse: Dienste / Dämonprozesse

Prozesserzeugung

Ein Prozess kann durch folgende Ereignisse erzeugt werden:

- Initialisierung des Systems
- Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
- Benutzeranfrage, einen neuen Prozess zu erzeugen
- Initiierung einer Stapelverarbeitung (Stapeljob)

Auf unixartigen Systemen werden Prozesse durch „Forking“ oder „Execution“ erzeugt. Die Bibliotheksfunktion `fork()` kopiert einen Prozess, so dass das alte Programm in zwei Prozessen ausgeführt wird. Die Funktion `exec*()` ersetzt in einem laufenden Prozess das alte Programm durch ein neues.

Prozesshierarchie

Ein Prozess, der einen anderen Prozess erzeugt, wird Elternprozess genannt. Der neu erzeugte Prozess wird Kindprozess genannt. Der Kernel erstellt initial einen Ur-Prozess (unter UNIX oft `init` oder `systemd`). Dadurch ergibt sich eine Prozesshierarchie.

Systemaufrufe

Systemaufrufe sind Funktionsaufrufe eines Prozesses, die eine Funktion innerhalb des Kernels auslösen. Die übergebenen Werte werden dabei in den Kernelspace kopiert und die Resultate kommen als Rückgabewerte zurück in den Userspace. Auf diese Weise können Aktionen durchgeführt werden, die auf den Kernel zurückfallen (bspw. das Erzeugen und Beenden von Prozessen).

Copy-on-Write

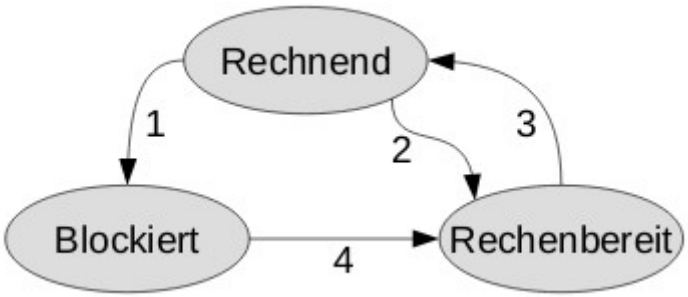
Jeder Prozess erhält seinen eigenen Speicherbereich. Es wird aber effizienter Weise erst dann ein eigener Speicherbereich kopiert, wenn er überschrieben wird. Eltern- und Kindprozesse teilen sich somit den selben Speicher. Erst wenn einzelne Werte im Speicher überschrieben werden, erhält der Kindprozess seinen eigenen Speicherbereich.

Prozessbeendigung

Das Ende eines Prozesses kann durch unterschiedliche Ursachen eingeleitet werden:

- Normales Beenden (freiwillig): Der Programmcode ist am Ende angekommen, der Prozess ist fertig und beendet sich
- Beenden aufgrund eines Fehlers (freiwillig): Der Prozess hat einen Fehler selbst erkannt und beendet sich freiwillig (z.B. wenn er eine angegebene Datei nicht finden kann)
- Beenden aufgrund eines schwerwiegenden Fehlers (unfreiwillig): Der Prozess kann wegen eines unvorhergesehenen Fehlers nicht weiter arbeiten (z.B. Nulldivision)
- Beenden durch einen anderen Prozess (unfreiwillig): Ein anderer Prozess zwingt den Prozess sich zu beenden (z.B. durch das Signal SIGKILL)

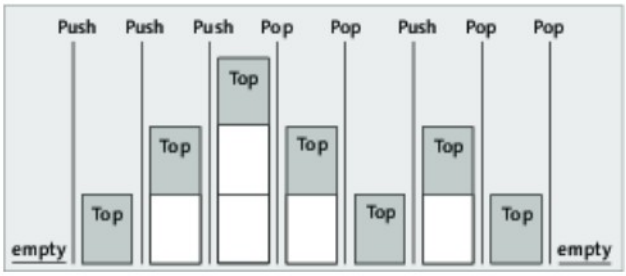
Prozesszustände



1. Prozess ist blockiert, weil er auf Eingabe wartet
2. Scheduler wählt einen anderen Prozess aus
3. Scheduler wählt diesen Prozess aus
4. Eingabe vorhanden

Stack

Jeder Prozess verfügt über einen Stack. Bei Funktionsaufrufen werden Rücksprungadresse (Befehlszeiger), Funktionsparameter und Rückgabewerte mit PUSH auf den Stack gelegt und mit POP wieder entfernt. Es handelt sich um ein LIFO-Prinzip. Zum Beginn und zum Ende eines Prozesses ist sein Stack „leer“.



Threads

Threads (auch Miniprozesse oder leichtgewichtige Prozesse genannt) sind Ausführungsstränge eines Prozesses. Sie erlauben die parallele Ausführung von Codebestandteilen in einem gemeinsamen Speicherbereich. Die Parallelisierung erfolgt dabei innerhalb eines Prozesses und nicht durch mehrere parallel ausgeführte Prozesse.

Beim Vergleich vom Prozessen und Threads sind folgende Aspekte zentral:

- zwei Prozesse verwenden unterschiedliche Adressräume, Threads hingegen teilen sich ihren gemeinsamen Adressraum
- der Overhead für die Kommunikation zwischen Prozessen kann reduziert werden, wenn stattdessen Threads verwendet werden
- Threads sind an einen Prozess gebunden. Das Ende des Prozesses bedeutet auch das Ende aller seiner Threads. Prozesse hingegen sind weitestgehend voneinander unabhängig
- Da Threads keinen eigenen Speicher und weniger Verwaltungsoverhead erzeugen, können sie schneller erzeugt und beendet werden
- Threads werden eher zur Performancesteigerung innerhalb von Prozessen verwendet, als zur Performancesteigerung des Gesamtsystems

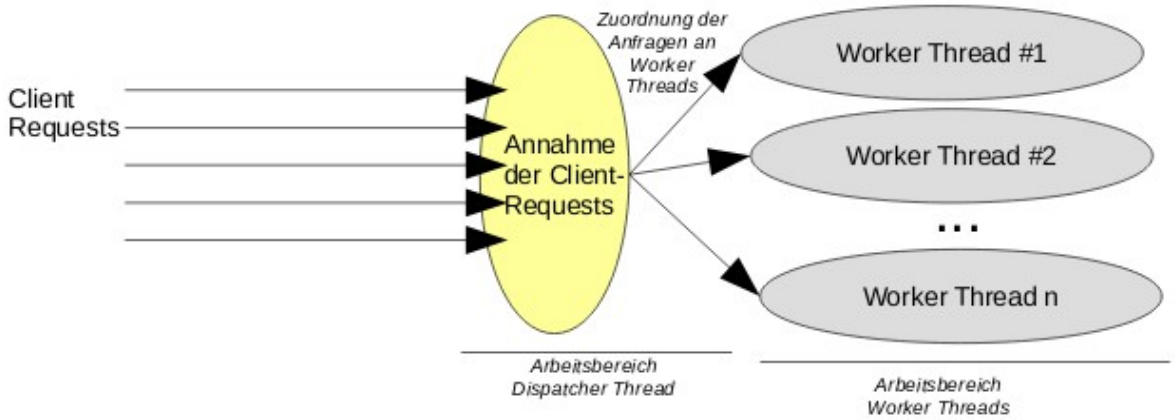
Multithreading

Die parallele Ausführung mehrerer Threads wird Multithreading genannt. Zwischen den Threads findet ein schneller Wechsel mit kurzen Zeitfenstern statt. Wie auch Prozesse können Threads unterschiedliche Zustände (rechnend, rechenbereit, blockiert) annehmen.

Dabei erhält jeder Thread seinen eigenen Stack und virtuellen Befehlszeiger. Globale Variablen sind threadübergreifend zugreifbar. Geöffnete Dateien und Signale sind ebenfalls für alle Threads zugreifbar.

Typische Thread-Architektur

Eine simple Aufteilung eines Prozesses in Threads besteht aus einem Dispatcher-Thread, der mehrere Worker-Threads verwaltet:



Interprozesskommunikation

Aufgaben der Interprozesskommunikation

Die Interprozesskommunikation dient dem Austausch von Daten zwischen Prozessen. Es muss sichergestellt werden, dass zwei Prozesse miteinander kommunizieren können, mehrere Prozesse nicht gleichzeitig auf dieselbe Ressource zugreifen und dass es einen sauberen Ablauf bei Abhängigkeiten zwischen Prozessen gibt.

Techniken der Interprozesskommunikation

- **Pipes und FIFOs** sind eine Methode zur Kommunikation zwischen ein bis zwei Prozessen über Dateideskriptoren. Pipes erlauben den Austausch von Daten über den Kernel mit dem Syscall `pipe()`. FIFOs sind Pipes, die als Dateien im System abgelegt werden. Sie können mit dem Syscall `mkfifo()` erstellt werden. Während Pipes nach Beendigung der Prozesse geschlossen werden, können FIFOs weiterhin bestehen. In eine FIFO können auch mehrere Prozesse gleichzeitig schreiben.
- Mit **Shared Memory** gibt es die Möglichkeit, einen Speicherbereich zwischen mehreren Prozessen zu teilen. Durch `shm_open()` kann ein geteilter Speicherbereich erzeugt werden, der über das dafür erzeugte Handle in den lokalen Prozess und in fremde Prozesse gemappt werden kann.
- Wenn man nicht möchte, dass zwei Prozesse gleichzeitig auf den Speicher schreiben, kann man **Message Queues** verwenden. Sie stellen eine Warteschlange dar. Mit ihnen werden Nachrichten eines bestimmten Typs gesendet, die dann nacheinander vom Empfänger abgeholt werden.
- **Sockets** sind Handles, die meist durch einen 4er-Tupel (Sendeadresse, Zieladresse, Sendeport, Zielport) identifiziert werden. Sie erlauben eine bidirektionale Kommunikation zwischen lokalen Prozessen bzw. zwischen Netzwerkprozessen. Sockets müssen miteinander verbunden und ggf. wieder abgebaut werden.
- Prozesse können sich gegenseitig **Signale** schicken um zu kommunizieren. Mithilfe des Syscalls `kill(pid, signum)` wird ein Signal gesendet. Dabei gibt `pid` die ID des Zielprozesses an und `signum` das zu sendende Signal. Die empfangenen Signale werden prozessintern mit einem Signalhandler abgefangen. Dabei handelt sich um einen Callback, der über `signal(signum, handler)` festgelegt wird. Der Parameter `handler` ist dabei ein Funktionszeiger auf den Callback, der ausgelöst wird, wenn das Signal empfangen wird.

Race Conditions

Eine Race Condition (ein Wettrennen um den Zugriff auf Ressourcen) ist eine Konstellation, in der das Ergebnis einer Operation vom zeitlichen Verhalten bestimmter Einzeloperationen abhängt. Race Conditions können die Verfügbarkeit von Prozessen beeinflussen und die Vertraulichkeit von Daten beeinträchtigen.

Kritische Region und Lösungen

Als kritische Region wird der Teil eines Codes bezeichnet, der von mehreren Prozessen verwendet wird und eine potentielle Race Condition darstellt. Um den gleichzeitigen Zugriff auf Dateien zu verhindern, gibt es folgende Lösungen:

- **File Locks:** Die Ressource wird von einem Prozess gesperrt, sodass nur eben dieser Prozess auf sie zugreifen kann, bis er die Ressource wieder entsperrt. Die durchgeführten Operationen werden somit atomar.
- **Temporäre Dateien:** Es wird eine temporäre Datei mit einzigartigem Namen erstellt, sodass andere Prozesse nicht zufällig denselben Dateinamen verwenden können. Die Datei wird ebenfalls gesperrt und nach dem Schreibvorgang in den gewünschten Dateinamen umbenannt.
- **Zugriffsrechte:** Temporärdateien können nach dem Erstellen von anderen Prozessen gesichtet und beschrieben werden, bevor der Lock gesetzt ist. Um dies zu verhindern, kann man direkt bei der Erstellung der Datei exklusive Zugriffsrechte setzen.

Folgende Aspekte sind für die Behandlung kritischer Regionen wichtig:

- Es dürfen keine zwei Prozesse gleichzeitig in ihren kritischen Regionen sein
- Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der CPUs gemacht werden
- Kein Prozess, der außerhalb seiner kritischen Region läuft, darf einen anderen blockieren
- Kein Prozess sollte ewig darauf warten müssen, in seine kritische Region einzutreten

Semaphoren und Mutexe

Mit Semaphoren kann festgestellt werden, ob derzeit ein anderer Prozess auf eine Ressource zugreift. Es handelt sich um einen Counter, der die Anzahl der Zugriffe repräsentiert. Die Bearbeitung eines Semaphors ist atomar.

Ein binärer Semaphore, der nur die Information liefert, ob ein kritischer Bereich derzeit betreten wird oder nicht, der jedoch keinen Counter besitzt, heißt Mutex.

Erzeuger-Verbraucher-Problem

Beim Erzeuger-Verbraucher-Problem gibt es einen Erzeuger- und einen Verbraucherprozess. Beide verwenden einen gemeinsamen Puffer mit fester Größe und arbeiten gleichzeitig. Wenn der Puffer voll wird, legt sich der Erzeuger schlafen und wenn der Puffer leer wird, legt sich der Verbraucher schlafen. Ein Prozess kann sich nicht selbst wecken und muss deswegen vom jeweils anderen wieder geweckt werden.

```
int N = 100, count = 0

producer():
    int item
    while TRUE:
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);

consumer():
    int item
    while TRUE:
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
```

Während ein Prozess die Variable `count` prüft, kann er vom Scheduler unterbrochen werden und der andere Prozess die Variable verändern. Somit kommt es zu einer Race Condition mit dem Resultat, dass beide Prozesse für immer schlafen.

Das Problem kann durch die Verwendung von Semaphoren und Mutexe gelöst werden. Das Eintreten in die kritische Region wird durch einen Mutex gesichert und der Zähler wird durch einen Semaphor ersetzt.

Schlafender-Friseur-Problem

In einem Friseursalon gibt es eine begrenzte Anzahl an Warteplätzen und einen Friseur, der sich schlafen legt, wenn kein Kunde da ist. Sobald ein Kunde den Raum betritt und sieht, dass der Friseur schläft, weckt er ihn. Wenn der Friseur gerade mit Haareschneiden beschäftigt ist, setzt sich der Kunde stattdessen in die Warteschlange und schläft, bis der Friseur Zeit für ihn hat und ihn weckt. Sollte ein Kunde kommen und sehen, dass alle Warteplätze belegt sind, geht er wieder.

Dabei kann es zu Race Conditions kommen: Der Friseur wird genau in dem Moment vom Scheduler unterbrochen, wenn er mit einem Kunden fertig wird und überprüft, ob ein weiterer Kunde wartet. Der Kunde, der gerade neu kommt, sieht, dass der Friseur noch wach ist und legt sich schlafen bis dieser ihn weckt. Der Friseur, der nun wieder Rechenzeit bekommt, hat bereits überprüft, dass kein Kunde da ist, und legt sich nun ebenfalls schlafen. Somit schlafen beide.

Das Problem kann durch die Verwendung von Semaphoren und Mutexen gelöst werden.

Deadlocks

Als Deadlock wird eine Situation bezeichnet, in der ein System nicht mehr operieren kann, weil alle zugehörigen Prozesse sich gegenseitig schlafen gelegt haben und sich nicht mehr aufwecken können. Die folgenden Voraussetzungen müssen erfüllt sein, damit ein Deadlock entstehen kann:

- Jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet.
- Prozesse, die schon Ressourcen reserviert haben, können noch weitere anfordern.
- Ressourcen, die einem Prozess bewilligt wurden, können diesem nicht gewaltsam wieder entzogen werden. Der Prozess muss sie explizit freigeben.
- Es gibt eine zyklische Liste von Prozessen, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess gehört.

Deadlocks können durch die korrekte Verwendung von Semaphoren und / oder Mutexen verhindert werden.

Barrieren

Manchmal kann eine Software erst eine neue Aufgabe ausführen,wenn mehrere Threads oder Prozesse eine bestimmte Aufgabe erledigt haben. Das heißt, wenn alle dafür notwendigen Threads bzw. Prozesse beendet sind.

Barrieren lösen dieses Problem, indem alle relevanten Threads und Prozesse zuerst bei der Barriere ihre Vollendung melden müssen, bevor sie weiterlaufen dürfen. Die Barriere gibt das Weiterlaufen dieser Threads bzw. Prozesse erst dann frei, wenn tatsächlich alle notwendigen Aufgaben erledigt sind.

Scheduling

Scheduler

Der Scheduler ist der Teil eines Betriebssystems, der entscheidet, wann und wie lange Prozesse Rechenzeit zugeteilt bekommen. Somit legt der Scheduler fest, welcher Prozess gerade auf welchem CPU-Kern ausgeführt wird. Dabei gibt es verschiedene Schedulingalgorithmen, die angewendet werden können. Gibt es gerade gar nichts zu tun, läuft ein Idleprozess.

Scheduling ist aus den folgenden Gründen nicht trivial:

- Prozesse können unterschiedlich viel Rechenzeit benötigen und erhalten
- Prozesse können unterschiedliche Prioritäten aufweisen
- Interaktive Prozesse benötigen eine besonders zeitnahe Bearbeitung
- Echtzeitprozesse müssen evtl. gesondert berücksichtigt werden
- Es muss geklärt werden in welcher Frequenz ein Prozesswechsel stattfindet
- Threads und Prozesse müssen evtl. unterschiedlich behandelt werden

Optimierungsproblem

Der Scheduler selbst ist ein Teil des Kernels und benötigt Rechenzeit, um seinen Algorithmus durchzuführen. Je aufwendiger der Schedulingalgorithmus, umso weniger Rechenzeit steht den Anwendungen zur Verfügung.

Weiterhin ist zu beachten, dass der Scheduler bei jedem Prozesswechsel aufgerufen werden muss. Das heißt, je öfter wir Prozesse voneinander ablösen, umso öfter muss auch der Scheduler laufen, womit ein gewisser Overhead erzeugt wird.

Context Switching

Um von einem bisher laufenden Prozess zu einem neuen Prozess zu wechseln, ist ein sogenannter Context Switch notwendig. Dieser besteht aus den folgenden Schritten:

- Der bisherige Prozess lief im User Mode. Das Betriebssystem muss zunächst in den Kernel Mode wechseln (Mode Switch).
- Im Kernel Mode kann der Zustand des aktuellen Prozesses (Stack, Register, Befehlszeiger, ...) gespeichert werden.
- Der Scheduler läuft ebenfalls im Kernel Mode und wählt nun den Prozess aus, der als nächstes Rechenzeit erhalten soll.
- Die gespeicherten Daten des ausgewählten Prozesses werden geladen und der Prozess kann im User Mode weiterlaufen.

Nicht-unterbrechendes Scheduling (Cooperative Multitasking)

Ein nicht-unterbrechender (non-preemptive) Scheduler weist einem Prozess Rechenzeit zu. Der Prozess läuft so lange, bis er blockiert oder freiwillig die CPU freigibt. Theoretisch darf der Prozess unbegrenzt lange laufen.

Unterbrechendes Scheduling

Beim unterbrechenden (preemptive) Scheduling wird ein Prozess vom Scheduler unterbrochen. Der Scheduler steuert, welcher Prozess wann Rechenzeit erhält und für wie lange. Die Steuerung der Zuteilung erfolgt mit Timern und Interrupts.

Ziele von Schedulingalgorithmen

- Fairness: faire Zuteilung von Rechenzeit an Prozesse
- Policy Enforcement: Umsetzung von Vorgaben (z.B. Zeit für Sicherheitsüberprüfung)
- Balance: möglichst gleichmäßige Auslastung aller Komponenten

Scheduling für Stapelverarbeitung

Das Ziel eines Schedulers für Stapelverarbeitungssysteme ist die Maximierung des Durchsatzes (Jobs pro Stunde), die Minimierung der Durchlaufzeit und die Maximierung der CPU-Auslastung. Es gibt folgende Algorithmen:

- Bei **First-Come-Frist-Served** laufen die Jobs in der Reihenfolge, in der sie Rechenzeit anfordern, so lange sie wollen. Dieser Algorithmus ist einfach zu verstehen und zu implementieren. Es handelt sich um eine simple FIFO-Warteschlange. Der Nachteil ist, dass Jobs mit langer Rechenzeit andere Jobs mit kurzer Rechenzeit blockieren können und somit der Durchsatz nicht optimal ist.
- Der Algorithmus **Shortest-Job-First** gibt immer dem Prozess Rechenzeit, der die kürzeste Laufzeit hat. Voraussetzung ist, dass alle Laufzeiten der zu planenden Prozesse bekannt sind. Nach der Beendigung jedes vollendeten Jobs wird der Algorithmus erneut ausgeführt. Der Vorteil ist, dass dadurch die durchschnittliche Wartezeit minimiert ist. Der Nachteil ist, dass Jobs mit langer Laufzeit besonders lange auf Rechenzeit warten müssen.

Scheduling für interaktive Systeme

Bei interaktiven Systemen sollte die Antwortzeit minimal sein, insbesondere wenn ein Nutzer auf Feedback wartet. Zudem sollten die Erwartungen zur Reaktionszeit des Benutzer beachtet werden (Proportionalität). Es gibt folgende Algorithmen:

- Bei **Round Robin** wird die Prozessliste in ihrer gegebenen Reihenfolge abgearbeitet. Sind alle Prozesse an der Reihe gewesen, wird wieder von vorne begonnen. Jeder Prozess erhält dabei dasselbe Quantum (Zeitfenster) an Rechenzeit. Der Algorithmus ist leicht zu verstehen und implementieren, allerdings erzeugt die Gleichbehandlung aller Prozesse viele Kontextwechsel, was zu einem hohen Overhead führen kann.
- Der **Priority-Scheduling**-Algorithmus erhalten die Prozesse unterschiedliche Prioritäten und Prozesse mit höherer Priorität erhalten mehr Rechenzeit. Der Vorteil ist, dass wichtige Prozesse bevorzugt behandelt werden und Prioritäten dynamisch zugewiesen werden können. Das Problem ist allerdings, dass optimale Prioritäten schwer zu finden sind und bspw. speicherintensive Prozesse mit geringer Priorität sehr lange Ressourcen belegen.
- Bei **Shortest-Process-Next** wird die Minimierung der Wartezeit erreicht, indem die benötigten Laufzeiten der Prozesse geschätzt werden. Dies geschieht auf Basis von Messungen der bisherigen Laufzeiten. Die Messungen werden dabei nach Alter gewichtet und entsprechend neue Zeitfenster vergeben. Das Vorgehen erfolgt dabei rundenweise mit einer Gewichtung α, sodass wir $\alpha T_0 + (1-\alpha) T_1$ an Zeit in einer neuen Runde zuweisen. Das Problem ist, dass es durch das Cachen und Berechnen der Laufzeiten zu viel Overhead kommt.

- Beim **Lottery-Scheduling** erhalten die Prozesse Lose durch die sie Rechenzeit gewinnen können. Prozesse mit höherer Priorität können mehrere Lose erhalten. Der Anteil f an Gesamtlosen führt zu etwa f% der Rechenzeit. Es handelt sich dabei um Cooperative Scheduling. Ein Problem besteht allerdings darin, festzulegen, wie viel Lose ein Prozess nun bekommen kann. Dies kann nur durch den Programmierer festgelegt werden.
- Der **Fair-Share**-Algorithmus teilt die Rechenzeit nicht zwischen den Prozessen sondern zwischen den Benutzern auf. Es gibt auch die Möglichkeit, Benutzer zu priorisieren. Bei diesem Algorithmus ist es situationsabhängig, ob er als fair bezeichnet werden kann und er ist nicht auf allen interaktiven Systemen praktikabel.
- Beim **Multilevel-Feedback-Queue**-Algorithmus gibt es verschiedene Queues mit unterschiedlich langen Zuweisungen von Rechenzeit. Ein neuer Prozess startet in der Top-Level-Queue. Wird er innerhalb eines bestimmten Quantums beendet, wird er aus der Queue entfernt. Ansonsten wird er in die nächst-niedrigere Queue überführt und bekommt dort mehr Rechenzeit zugeteilt. So kann ein Prozess, der nicht beendet, durch mehrere Queues wandern, bis er in der Bottom-Level-Queue ankommt. Dieser Algorithmus hat den Vorteil, dass er stark parametrisierbar ist (Anzahl der Queues, Wahl des Algorithmus pro Queue, Algorithmus zum Wechseln der Queues, …).

Scheduling für Echtzeitsysteme

Scheduler für Echtzeitsysteme haben das Ziel, Deadlines einzuhalten. Man unterscheidet zwischen harten und weichen Echtzeitsystemen. Bei harten Echtzeitsystemen müssen Deadlines eingehalten werden und bei weichen ist es zumindest unter Einbußen auch tolerierbar, eine Deadline nicht einzuhalten.

Ein Ereignis i tritt mit einer Periode P_i auf und benötigt C_i Sekunden Rechenzeit. Das System gilt bei m zu bedienenden Echtzeitereignissen dann als scheduable, wenn gilt:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Der **Earliest-Deadline-Frist**-Algorithmus sortiert die Prozesse aufsteigend nach ihren Deadlines. Der Prozess mit der dringendsten Deadline wird zuerst ausgeführt.

Scheduling für Threads

Threads können vom Betriebssystem wie Prozesse behandelt werden. Unter Linux spricht man daher auch von Tasks. Ein Task kann ein Prozess oder ein Thread sein. Dabei werden Threads analog zu Prozessen im Kernel abgebildet.

Andere Betriebssysteme implementieren Threads wiederum im Userspace. Der Scheduler selektiert dann nur den Prozess, der Rechenzeit erhalten soll. Anschließend übernimmt der prozessinterne Scheduler das Scheduling der Threads.

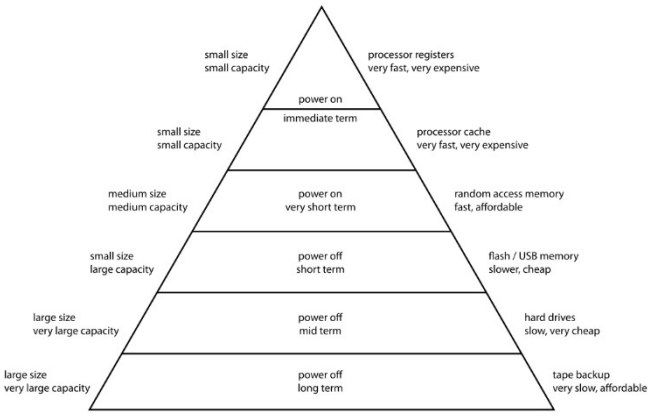
Der Threadwechsel im Userspace kann ohne Context Switch erfolgen und ist somit performanter.

Speicherverwaltung

Warum Speicherverwaltung?

Jeder Programmierer und Benutzer hätte am liebsten für alle Programme den gesamten Speicher eines Systems zu Verfügung. Zudem hätten sie am liebsten nur nicht-flüchtigen Speicher mit der höchst möglichsten Performance.

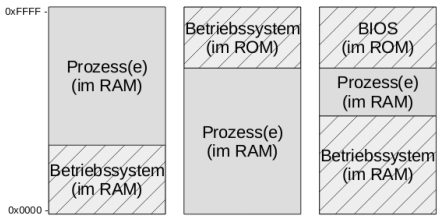
In der Realität müssen sich aber alle Programme den verfügbaren Speicher teilen. Zudem ist ein Teil des Speichers flüchtig oder relativ langsam. Der Speicherbedarf von Prozessen ändert sich ständig. Aus diesem Grund bedarf es eines ausgefeilten Speichermanagements.



Systeme ohne Speicherabstraktion

Frühe Großrechner setzten keine sogenannte Speicherabstraktion ein. Das heißt, dass jeder Prozess direkt auf dem physikalischen Speicher zugreifen konnte. Daraus resultiert das Problem, dass Programme gegenseitig ihre Werte lesen und überschreiben konnten. Entsprechend war es praktisch nicht möglich, zwei Prozesse parallel ablaufen zu lassen.

Je nach System wird der physikalische Speicher bei fehlender Abstraktion zwischen Benutzerprogramm und Betriebssystem aufgeteilt. Dabei kann das Benutzerprogramm das Betriebssystem im schlimmsten Fall überschreiben, es sei denn, es befindet sich im ROM.



Auf einem solchem System könnten Programme nur dann „parallel“ ablaufen, wenn:

- entweder nur Threads innerhalb eines einzelnen Programms parallel ablaufen und sich diese Threads einen gemeinsamen Speicher teilen (dennoch Speicherverwaltung benötigt)
- oder wenn bei jedem Context Switch das aktuell laufende Programm komplett auf die Platte ausgelagert wird und beim nächsten Bearbeiten wieder komplett eingelesen wird (Overhead)

Speicherabstraktion

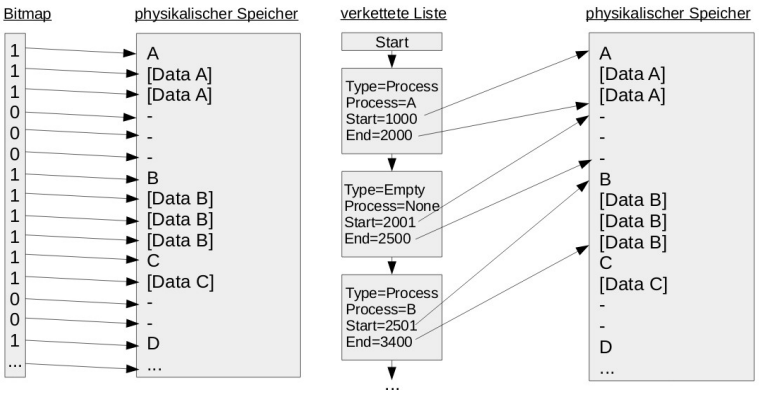
Die Speicherabstraktion basiert auf dem Konzept des Adressraums. Jeder Prozess erhält dabei seinen eigenen Adressraum und erhält auch nur auf diesen Zugriff, womit Prozesse gleichzeitig ablaufen können. Zwei Programme, die auf die gleiche Adresse (z.B. 0x00123) ihres Speicherbereichs zugreifen, werden in Wirklichkeit auf unterschiedliche Bereiche des physikalischen Speichers zugreifen.

Moderne Systeme arbeiten mit virtuellem Speicher. Dieser ist so groß wie das System theoretisch maximal adressieren könnte unabhängig von der tatsächlichen Größe des Hauptspeichers. Bei einem 64-Bit-Betriebssystem wären das 2^{64} Bit. Somit kann bspw. ein System mit 4 GiB Speicher ein Programm ausführen, das 16 GiB benötigt. Die virtuellen Adressen der Prozesse müssen auf die physikalischen Adressen gemappt werden.

Grundtechniken der Speicherabstraktion

Es gibt zwei Grundtechniken zur Speicherverwaltung: Bitmaps und verkettete Listen. Bitmaps speichern lediglich für jeden Speicherbereich ob dieser belegt (1) oder frei (0) ist. Es wir für jeden Speicherbereich nur ein Bit benötigt.

Eine verkette Liste hingehen merkt sich zusammenhängende Speicherbereiche (jeweils von Anfang bis Ende) sowie deren Status. Der Speicherbedarf einer verketteten Liste ist jedoch größer, da zusätzlich Metadaten gespeichert werden.



Algorithmen für Speicherallokation

Wenn ein Prozess neuen Speicher anfordert, muss das Betriebssystem eine optimale Stelle im Hauptspeicher finden. Zur Allokation gib es folgende Algorithmen:

- **First Fit**: Die freien Speicherbereiche werden durchgegangen. Der erste Speicherbereich, der groß genug für die Allokation ist, wird verwendet. Der Algorithmus ist schnell, weil er wenig sucht, erzeugt aber ggf. große Lücken.
- **Next Fit**: Funktioniert wie First Fit, merkt sich allerdings die zuletzt benutzte Speicherstelle und sucht von dieser ausgehend. Simulationen zeigten allerdings schlechtere Performance als First Fit.
- **Best Fit**: Durchsucht die ganze Liste nach dem kleinsten Speicherbereich, der groß genug für die Allokation ist. Der Algorithmus ist langsam, weil er lange suchen muss, und er erzeugt viele winzige Lücken im Speicher, die anschließend nicht mehr verwendet werden können.
- **Worst Fit**: Durchsucht die ganze Liste nach dem größten Speicherbereich, damit die verbliebenen Lücken noch genutzt werden können. Der Algorithmus ist ebenfalls wegen der Suche langsam und Simulationen zeigten keine optimalen Ergebnisse.
- **Quick Fit**: Der Algorithmus verwaltet mehrere Listen, die jeweils auf Speicherbereiche in bestimmten Größenbereichen verweisen (bspw. 16-31k und 32-64k). Der Algorithmus kann Lücken in vorgegebener Größe extrem schnell finden. Der Nachteil ist allerdings, dass bei Prozessterminierung Speicherbereiche zusammengelegt werden müssen, was aufwendig werden kann.

Paging

Im CPU-Gehäuse gibt eine Memory Management Unit (MMU), die virtuelle Adressen, die die CPU verarbeitet, in physikalische Adresse umwandelt und umgekehrt. Das heißt, die MMU ist für das Mapping der Adressen verantwortlich. Die Technik zur Übersetzung von virtuellen in physikalische Adressen wird **Paging** genannt.

Der virtuelle Adressraum ist in **Pages** (Seiten) unterteilt. Eine Page, die gerade auf den physikalischem Speicher abgebildet ist, heißt **Page Frame** (Seitenrahmen). Die Pages im virtuellen Speicher sind immer genauso groß wie die Page Frames im physikalischen Speicher.

Es wird ein Mechanismus benötigt, um sich zu merken, welche virtuellen Pages gerade im physikalischen Speicher geladen sind. Hierzu wird eine **Page Table** (Seitentabelle) benötigt, in der steht,

welche Page gerade auf welchem Page Frame abgebildet wurde. Diese Tabelle wird von der MMU verwaltet. Ist eine benötigte Page gerade nicht im physikalischen Speicher vorhanden, spricht man von einem **Page Fault**. In diesem Fall löst die MMU einen Syscall zum Laden der entsprechenden Page aus.

Page-Replacement-Algorithmen

Wenn ein Page Fault auftritt und es keinen freien Page Frame mehr im physikalischen Speicher gibt, muss ein anderer Page Frame ausgelagert werden. Dieser kann nicht willkürlich gewählt werden, da dies in der Regel zu einer sehr schlechten Performance führt. Daher wird ein Algorithmus zur Seitenersetzung benötigt.

Ein optimaler Algorithmus würde theoretisch immer wissen, welche Seite am längsten nicht benötigt wird. Allerdings kann das Betriebssystem das nicht im Voraus bestimmen und der Verwaltungsaufwand wäre extrem hoch.

- Beim **Not-Recently-Used**-Algorithmus (NRU) werden die Pages mit zwei verschiedenen Flags markiert: R (referenziert bzw. zugegriffen) und M (modifiziert). Diese Flags werden nach Ablauf einer Zeit durch einen Timer entfernt. Es ergeben sich vier Klassen von Markierungen:
  - Klasse 0: nicht referenziert, nicht modifiziert (!R, !M)
  - Klasse 1: nicht referenziert, modifiziert (!R, M)
  - Klasse 2: referenziert, nicht modifiziert (R, !M)
  - Klasse 3: referenziert, modifiziert (R, M)

  Der Algorithmus ersetzt nun immer Pages der möglichst geringsten Klasse. Referenzierungen werden dabei höher als Modifikationen bewerten. Dies führt nachweislich zu einer guten Performance.
- Der **FIFO**-Algorithmus verwaltet alle Pages in einer verketteten Liste. Neue Einträge werden an das Ende der Liste angefügt. Wenn eine Page ersetzt werden muss, wird immer die älteste Page vom Anfang der Liste entfernt. Problematisch ist dieser Algorithmus insbesondere deshalb, da nicht klar ist, ob die älteste Page noch verwendet wird.
- Der **Second-Chance**-Algorithmus ist eine Erweiterung von FIFO. Bei jedem Ersetzen wird das R-Flag geprüft. Wenn die Seite alt ist und das R-Flag nicht gesetzt ist, wird sie ersetzt. Hat die zu ersetzende Page allerdings den R-Flag gesetzt, wird stattdessen dieser entfernt und die Page kommt wieder an das Ende der Liste. Somit erhalten Pages mit gesetzten R-Flag eine zweite Chance.
- Der **Clock**-Algorithmus funktioniert wie Second-Chance mit einem R-Flag. Der Unterschied liegt darin, dass es sich um zyklisches Vorgehen handelt. Die Ersetzung erfolgt nicht nach dem Alter der Seite sondern gemäß dem Stand der Uhr.
- Der **Least-Recently-Used**-Algorithmus ähnelt dem optimalen Algorithmus. Es wird angenommen, dass eine Page, die besonders lange nicht mehr verwendet wurde, auch weiterhin mit einer geringeren Wahrscheinlichkeit benutzt wird, als eine, die erst kürzlich verwendet wurde. Dieser Algorithmus ist in der Praxis leider kaum implementierbar, da die Informationen aller Pages ständig aktualisiert und sortiert werden müssten.
- Der **Working-Set**-Algorithmus konzentriert sich auf die Menge von Seiten, mit denen ein Prozess momentan primär arbeitet (das sogenannte Working Set). Der Algorithmus merkt sich die zuletzt verwendeten Seiten eines Prozesses und wenn er wieder vom Scheduler geladen wird, werden gleich alle Pages aus dem Working Set für diesen Prozess geladen statt jede Page nach einem expliziten Page Fault zu laden (Demand Paging). Seiten werden hierbei folglich geladen, bevor sie verwendet werden. Dieses Prinzip nennt sich Prepaging.

Dateisysteme

Definition Dateisystem

Dateisysteme legen fest, wie Daten auf einem Medium gespeichert werden und wie die Speicherung der Daten organisiert wird. Sie ermöglichen das Auffinden von und Zugreifen auf Daten. Gespeichert werden Daten in Form von persistent logischen Informationseinheiten (Dateien).

Dateisysteme sind deshalb benötigt, da Prozesse während der Ausführung nicht unbedingt alle Daten im Adressraum ablegen können und weil Daten nach Prozessterminierung ggf. weiterhin benötigt werden. Weiterhin werden Dateisysteme benötigt, um mehrere Anwendungen auf dieselben Daten zugreifen zu lassen.

Anforderungen an Dateisysteme

- Sie müssen eine große Menge an Informationen speichern können.
- Sie müssen sicherstellen, dass gespeicherte Daten auch nach Beendigung eines auf die Daten zugreifenden Prozesses erhalten bleiben.
- Sie sollten auch auf Multitasking-Systemen einsetzbar sein.
- Sie müssen das Auffinden von abgelegten Informationen ermöglichen.
- Sie sollten Zugriffsrechte umsetzen.
- Sie müssen eine Liste der benutzten und unbenutzten Blöcke verwalten.

Dateinamen

Namenskonventionen hängen vom Betriebssystem ab. Unterschiedlich gehandhabt werden typischerweise Sonderzeichen, Whitespaces, Umlaute, Groß- und Kleinschreibung und die erlaubte Länge für Dateinamen. Die Dateiendung muss nicht zwingend dem Typ der Datei entsprechen.

Dateitypen

Auf UNIX-artigen Systemen existieren verschiedene Dateitypen:

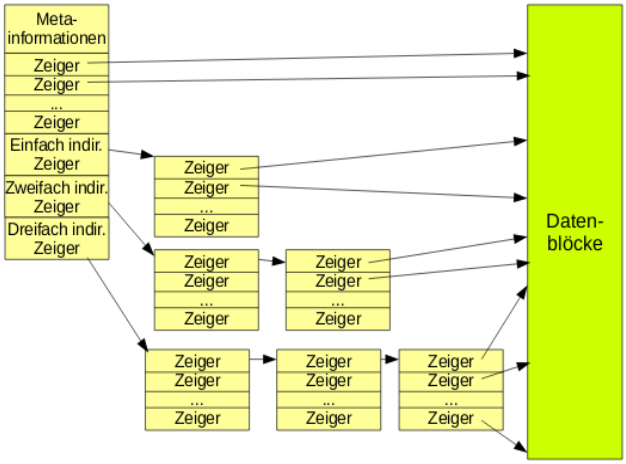
- reguläre Dateien
- Gerätedateien: Zeichendateien und Blockdateien
- Verzeichnisse
- Sockets
- Named Pipes
- Symbolische Links

Verzeichnisse

Verzeichnisse existieren nur in hierarchischen Dateisystemen, d.h. Dateisystemen, die Unterstrukturen bilden können. Verzeichnisdateien beinhalten eine Liste der in ihnen enthaltenen Dateien.

Inodes

Inodes stellen eine Hauptkomponente zur Dateiverwaltung unter den UNIX-Systemen dar. Jede Datei im Dateisystem verfügt über eine eigene Inode. Die Inode ist der Speicherort für alle Metadaten der Datei und auch für Verweise auf die Dateisystemblöcke, in denen die eigentlichen Dateiinhalte gespeichert sind.



Der Verweis auf die Blöcke der Datei funktioniert über Zeiger und (mehrfach) indirekte Zeiger. Die mehrfache Verwendung von indirekten Zeigern ist notwendig, damit die Größe der Datei dynamisch geändert werden kann und eine Fragmentierung möglich ist.

Eine Datei in einem Verzeichnis ist durch einen Namen sowie eine eindeutige Inode-Nummer gekennzeichnet. Über diese Inode-Nummer wird auf den jeweiligen Inode-Eintrag des Dateisystems zugegriffen, von welchem dann alle weiteren Informationen der Datei ausgelesen werden können. Der Dateiname selbst ist also nicht Bestandteil eines Inode-Eintrags, denn eine Datei kann mehrmals unter verschiedenen Namen – in Form von Hardlinks – im Dateisystem vorkommen. Der Kernel führt im Hintergrund, ohne dass ein Prozess etwas davon mitbekommt, mehrere Operationen durch, um eine Datei zu lesen. Der Kernel öffnet eine Datei bspw. durch den Syscall `open()`. Dabei muss die zugehörige Inode-Nummer der Datei ausgelesen werden. Mit dieser kann der Kernel auf den Inode-Eintrag der jeweiligen Datei und damit auf ihre Eigenschaften und den Dateiinhalt zugreifen.

Virtuelle Dateisysteme

Ein virtuelles Dateisystem abstrahiert Dateisystemspezifika und kann innerhalb eines Dateisystems eingebettet werden. So kann ein Wurzelverzeichnis bspw. Ext4 verwenden, an einem Punkt in der Verzeichnishierarchie aber auch eine virtuelles Dateisystem für eine DVD, ein Netzwerksystem oder ein USB-Stick einhängen. Für den Benutzer ist das jeweils eingesetzte Dateisystem transparent.

Journaling

Journaling zeichnet alle in einem Dateisystem zu bewerkstelligenden Veränderungen in einem Journal auf. Bei einem Stromausfall kann somit festgestellt werden, ob geplante Änderungen durchgeführt wurden oder nicht. Man unterscheidet zwischen zwei Arten von Journaling:

- **Meta Journaling**: Es werden nur Daten über Vorgänge gespeichert, aber keine Nutzdaten. Somit kann die Konsistenz der Verwaltungsinformationen sichergestellt werden.
- **Full Journaling**: Es werden ebenfalls die Originaldaten gespeichert, somit kann auch die Konsistenz der Nutzdaten sichergestellt werden.

Journaling ist aufwendig, hat sich aber in der ersten Variante in faktisch allen relevanten Betriebssystemen durchgesetzt. Es kann Bootvorgänge beschleunigen, indem Inkonsistenzen im Dateisystem schneller gefunden werden können.

IT-Sicherheit

Schutzziele

Es gibt drei wesentliche Schutzziele in der IT-Sicherheit:

- **Vertraulichkeit**: Es gibt keine unautorisierte Informationsgewinnung.
- **Integrität**: Es ist Subjekten nicht möglich, unautorisiert geschützte Daten zu manipulieren.
- **Verfügbarkeit**: Authentifizierte und autorisierte Subjekte werden in der Wahrnehmung ihrer Berechtigungen nicht unautorisiert beeinträchtigt.

Weitere Schutzziele für das Grundverständnis von IT-Sicherheit sind:

- **Authentizität**: Die Echtheit und Glaubwürdigkeit eines Objekts bzw. Subjekts sind gewährleistet.
- **Verbindlichkeit**: Die Verbindlichkeit einer Menge von Aktionen wird gewährleistet. Ein Subjekt kann im Nachhinein die Durchführung einer Aktion nicht abstreiten.
- **Anonymität**: Personenbezogene Daten können verändert werden, sodass Einzelangaben über persönliche oder sachliche Verhältnisse nicht mehr oder nur mit unverhältnismäßig großem Aufwand zugeordnet werden können.
- **Schwachstellen und Verwundbarkeit**: Eine Schwachstelle ist eine einem System inhärente Schwäche, die zu einer Verwundbarkeit führen kann. Eine Verwundbarkeit ist eine Schwachstelle, die ausnutzbar ist.
- **Bedrohung**: Möglichkeit, eine Verwundbarkeit auszunutzen, um Schutzziele zu verletzen
- **Risiko**: Wahrscheinlichkeit des Eintritts eines Schadensereignisses kombiniert mit der Höhe des potenziellen Schadens, der dadurch hervorgerufen werden kann.

Zugriffskontrollen

Eines der grundlegenden Probleme der IT-Sicherheit bzgl. Betriebssysteme ist der Schutz des Zugriffs auf Ressourcen. Es werden primär zwei Modelle unterschieden:

- **Discretionary Access Control**: Zugriff auf Daten wird anhand einer Identität festgelegt
- **Mandatory Access Control**: Zugriffskontrolle auf Basis von Regeln und Attributen

Verdeckter Kanal

Wird eine Sicherheitsrichtlinie verletzt, repräsentiert die zugehörige Kommunikation einen sogenannten verdeckten Kanal. Verdeckte Kanäle ...

- sind nicht im Systemdesign vorgesehen Kanäle,
- nutzen Fähigkeiten von Software, des Kernels oder der Hardware aus,
- umgehen eine Kommunikationsbarriere bzw. brechen eine Sicherheitsrichtlinie
- und existieren insbesondere innerhalb von lokalen Systemen und Netzwerken.

Sie betreffen potentiell jegliche Softwareentwicklung, bei der eine taskübergreifende Verbindung bestehen kann.

Bell-LaPadula-Modell

Das BLP-Modell sichert die Vertraulichkeit von Daten im Kontext eines Systems mit Multilevel-Security-Policy. Es enthält eine Menge von Sicherheitslevels. Eine höhere Klassifikation entspricht auch einer höheren Datensensitivität.

Jedes Subjekt hat eine **Clearance**, welche dem maximalen Sicherheitslevel des Subjekts entspricht. Jedes Objekt ist einer **Classification** zugeordnet. Clearance und Classification beziehen sich jeweils auf dieselben Levels. Der Unterschied besteht darin, dass ein Subjekt seinen Sicherheitslevel reduzieren kann, ein Objekt hingegen kann das nicht.

Darüber hinaus kann man **Kategorien** angeben, die spezifische Zugriffsberechtigungen auf Basis von bspw. Projekten oder Organisationseinheiten erlauben. Kategorien können sowohl Subjekten als auch Objekten zugewiesen werden. Die Kombination aus Clearance bzw. Classification und Kategorie wird **Compartment** genannt und folgendermaßen angegeben:

- Beispiel für Subjekt: `L(John) = (confidential, {accounting, support})`
- Beispiel für Objekt: `L(DateiX) = (secret, {accounting})`

Ein Compartment kann ein anderes dominieren. Dabei gilt folgende Regel:

- `(L, C) dom (L‘, C‘)` wenn `L‘ ≤ L und C‘ ⊆ C`

Für den Zugriff von Subjekten auf Objekte gelten nun folgende Regeln:

- **Lesezugriff** für ein Subjekt s auf ein Objekt o ist nur erlaubt, wenn gilt:
  `s dom o` und `L(s) ≥ L(o)`
- **Schreibzugriff** für ein Subjekt s auf ein Objekt o ist nur erlaubt, wenn gilt:
  `o dom s` und `L(s) ≤ L(o)`

Authentifikation

Um einen Benutzer zu identifizieren und benutzerbasierte Zugriffe zu gewähren, ist eine Authentifikation nötig.

- Ein Benutzer authentisiert sich gegenüber einem Betriebssystem (z.B. durch ein Passwort)
- Das Betriebssystem überprüft die Korrektheit und authentifiziert den Benutzer

Ein Passwort sollte jedoch nicht im Klartext gespeichert werden. Eine Hashfunktion speichert stattdessen eine Prüfsumme des Passworts. Es werden anschließend bei der Überprüfung des Passworts nur noch die Prüfsummen verglichen.

Zwei gleiche Passwörter würden dabei die gleiche Prüfsumme bilden. Deswegen wird zusätzlich ein sogenanntes Salt mit gespeichert, das mit in die Prüfsumme gerechnet wird. Dieses wird zufällig generiert und verhindert somit die gleiche Prüfsumme für gleiche Passwörter. Das Salt selbst ist im Klartext gespeichert.