BuS Panikzettel

"der Dude", "Walter Sobchak", Caspar Zecha, Philipp Schröer

5. August 2024

Inhaltsverzeichnis

Pseudonyme gehören anonymen Autoren, die anonym bleiben wollen.

1 Aufbau von Betriebssystemen

1.1 Kernel-/Usermode

Bei Betriebssystemen kann man zwischen zwei Modi unterscheiden, unter welchen Code laufen kann:

1. Kernel-Mode: Der Kernel-Mode ist ein privilegierter Modus, welcher unbeschränkten Zugriff auf alle Betriebsmittel zur Verfügung stellt. In ihm laufen die Kernel-Module.

2. User-Mode: In diesem restriktiven Modus laufen die Anwendungsprogramme. Wenn ein Systemprogramm Zugriff auf (in diesem Modus) beschränkte Betriebsmittel, braucht, so muss es mittels Syscalls auf dieses zugreifen.

1.2 Zwiebelschalenmodell

Im Zwiebelschalenmodell werden Zugriffsrechte nach Applikationen geordnetet. Im innersten Ring (0, mit vollem Zugriff) läuft der Kernel. In Ring 1 & 2 Gerätetreiber und im 3. Ring laufen die Anwendungsapplikationen.

In der Praxis wird einfach zwischen Kernel- und Usermode unterschieden, wobei Ring 1 und 2 dann ausgelassen werden.

1.3 Verschiedene Kernelmodelle

Man unterscheidet im Allgemeinen zwischen 3 verschiedenen Kernelmodellen:

1. Monolithischer Kernel: In einem Monolithischen Kernel sind nicht nur Funktionen zur Verwaltung von Betriebssystemen direkt eingebaut, sondern auch Gerätetreiber und eventuell weitere Funktionen. Er ist relativ Fehleranfällig, da alle BS-Komponenten im priviligierten Modus laufen. Beispiel: Linux

2. Mikrokernel: Ein Mikrokernel verfügt dagegen nur über grundlegende Funktionalitäten. Weitere Funktionalitäten werden in den Usermode ausgelagert. Er ist somit schlank, aber langsam, da oft zwischen User- und Kernelmode gewechselt werden muss. Beispiel: Minix

3. Hybridkernel: Ein Hybrid-Kernel ist eine Mischung beider oben genannten Modelle, und daher eine "gesunde"Mischung aus Geschwindigkeit und Stabilität. Dabei sind einige performancekritische Teile im Kernel-Modus, andere hingegen in eigenen Prozessen im User-Mode. Beispiel: Windows

2 Die Bash

Bash-Programmierung muss zum ordentlichen Verständnis selber geübt werden.

2.1 Übliche Struktur von Befehlsaufrufen

Programme werden auf der Bash überlicherweise wie folgt aufgerufen:
`$ command -option -option2 arg1 arg2`. (Das Dollar-Zeichen gehört nicht zum Kommando)

Beispiel:
`$ ls -l /var`
Gibt den Verzeichnisinhalt von dem Argument (`/var`) als Liste aus (Option `-l`).

2.2 Pipes

Mit Pipes lassen sich die Ein- und Ausgaben von Programmen umleiten.

Zeichen	*Erklärung*	*Beispiel*	*Erklärung des Beispiels*
|	Leitet die Ausgabe eines Programmes in die Eingabe des nächsten um	`echo "Test"| wc`	Führt word count für die Zeichenkette "Test" aus
>	Schreibt in Datei	`echo "Test"> a`	In der Datei a steht die Zeichenkette "Test"
<	Liest aus Datei	`wc < a`	Ruft word count für den Inhalt der Datei a auf
>>	Wie > nur dass an die Datei angehangen wird	`echo "Test">> a`	In der Datei a wird die Zeichenkette "Test" angehangen

2.3 Ein-/Ausgabe Streams

Jedes Kommando verfügt über drei Kanäle

- 0 - die Standardeingabe (Tastatur)
- 1 - die Standardausgabe (Bildschirm)
- 2 - Fehlerkanal (Bildschirm)

Kanäle können auch umgeleitet werden, bspsw. `$ (echo "Test2" 1 > &2) 2> a`. Dies leitet die Zeichenkette "Test"von der Standardausgabe (1) auf die Fehlerausgabe (referenziert durch & 2) und leitet die Fehlerausgabe (2) in die Datei a.

2.4 Variablen

In einem Bash-Skript kann man auf Parameter mit `$1`, `$2`, `$3` usw. zugreifen.

2.5 Kommandosubstitution

Im Kommando `echo $(ls)` wird zuerst `ls` aufgerufen und die Ausgabe dann als Parameter in `echo` eingesetzt.

2.6 Bedingungen, Schleifen

2.6.1 If-Bedingungen

Syntax: `if [` *Bedingung* `]; then` *Anweisungen*`; fi;`
Beispiel:

```
if [ $1 = $2 ]; then
        echo "Die beiden Argumente sind gleich"
    else
        echo "Die beiden Argumente sind nicht gleich"
fi
```

2.6.2 For-Schleifen

Syntax: `for` *Variable* `in` *Liste* `do` *Anweisungen*`; done;`
Beispiel:

```
for i in "Test"
do
        echo $i
done
```

2.6.3 While/Until-Schleifen

Syntax: `while [` *Bedingung* `] do` *Anweisungen*`; done;`
`until [` *Bedingung* `] do` *Anweisungen*`; done;`

3 C

C verwendet im Gegensatz zu Java manuelle Speicherverwaltung. Dabei muss man besonders aufpassen.

3.1 Normale Pointer

Pointer finden in C rege Verwendung, so ist z.B. jedes Array im Prinzip ein Pointer auf die erste Position des Arrays. Pointer sind Variablen, welche Speicheradressen (z.B. von

anderen Variablen oder von Funktionen) beinhalten. Dereferenzieren (also auf die hinterlegte Speicheradresse zugreifen) kann man einen Pointer durch den Sternchenoperator *. Die Adresse einer Variable erhält man in C durch den &-Operator. Beispiel:

```
int a = 5;
int *b = &a;
int c = *b;
```

Hier besitzt die Integer-Variable **a** den Wert 5, **b** beinhaltet die Speicheradresse der Variable **a** und **c** ist eine Integer-Variable mit dem Wert 5.

3.2 Funktionspointer

Wie oben schon erwähnt sind in C auch Pointer auf Funktionen möglich, welche meist verwendet werden um z.B. Funktionen als Argument an eine andere Funktion zu übergeben. Beispiel:

```
int *func (int d) { ... }

int * (*funcptr)(int d) = &func;
```

Hier ist `funcptr` ein Pointer auf die Funktion `func` mit dem Rückgabetyp `int*`

3.3 Speicherverwaltung

C unterscheidet in der Speicherverwaltung zwischen dem Heap und dem Stack. Variablen werden normalerweise auf dem Stack alloziert (z.B. `int a = 5;`). Dem gegenüber steht der Heap. Alloziert wird hier mit `void* malloc(size_t)` jedoch muss der Speicher auch wieder mit `free(void*)` freigegeben werden. Beispiel:

```
int *a = malloc(sizeof(int)*2);
a[0] = 1; a[1] = 2;
free(a);
```

3.4 structs

Mit einem `struct` lassen sich Datentypen abstrahieren. Beispiel:

```
struct Point {
        int x; int y;
};
```

Auf die Member eines `struct` wird mittels des Punktoperator zugegriffen: `point.x = 5` oder falls `point` ein Zeiger auf ein `struct` ist, kann man sich mittels des `->` Operators das Dereferenzieren sparen: `point->x = 5`

4 Prozessverwaltung

4.1 Process Control Block (PCB)

In dem PCB (in Linux implementiert als `struct task_struct`) speichert das Betriebssystem alle relevanten Informationen eines Prozesses. So enthält ein PCB z.B.:

- PID (Process ID)
- Prozesszustand (ready, running, waiting, terminated)
- Informationen zum virtuellen Speicher
- Verwendete Dateien
- Signalbehandlungsroutinen
- Threadinformationen
- ...

4.2 Interprozesskomunikation

4.2.1 Pipes

Es wird mittels `pipe(int fd[2])` eine neue Pipe angelegt. Dabei ist `fd[0]` der Leseingang und `fd[1]` der Schreibeingang. Geschrieben/Gelesen werden kann auf Pipes wie bei Dateien. Der Syscall `dup2(int fd1, int fd2)` verbindet zwei Pipes.

4.2.2 Named-Pipes

Named-Pipes sind nichts anderes als Dateien, welche Pipes repräsentieren. Erstellt werden diese mittels `$ mkfifo test_pipe`.

4.2.3 Message-Passing/Message-Queueing

Bei diesem Konzept werden Nachrichten an eine Queue oder einen Empfänger (mittels `send`) gesendet und mittels (`receive`) empfangen.

4.2.4 Shared-Memory

Beim Konzept des Shared-Memorys haben alle Prozesse Zugriff auf einen gemeinsamen Speicher um sich hierüber auszutauschen. Da Zugriffoperationen auf den Speicher jedoch nicht atomar sind, ist hier Vorsicht geboten, und man sollte die auf den Speicher zugreifenden Prozesse *synchronisieren*.

4.3 Threads

Threads sind einzelne unabhängige Kontrollflüsse innerhalb eines Prozesses. Da Threads mehr Kontext als Prozesse teilen, ist unter anderem auch der Kontextwechsel schneller und Kommunikation einfacher. Threads teilen sich untereinander:

- den Adressraum des zugrundliegenden Prozesses
- globale Variablen und geöffnete Dateien
- die CPU-Zeit

Jedoch hat jeder Thread:

- einen eigenen Stack
- einen eigenen Program Counter (PC)
- einen eigenen Status
- eigene Registerwerte

Unter *nix-Betriebssysten wird ein Thread mit dem Syscall `clone()` (ähnlich zu `fork`) erstellt. Meistens wird dabei die PThreads-Implementierung verwendet.

4.3.1 Threadkonzepte

Es gibt zwei grundlegende Thread-Konzepte:

- User-Threads
  - Thread-Management in der Applikation (mittels Libraries)
  - Kernel sieht nur den Prozess
  - Geringer Aufwand für den Wechsel zwischen Threads
  - Blockiert ein Thread blockiert der komplette Prozess
- Kernel-Threads
  - Thread-Management durch den Kernel
  - Nötig für Multicore-Berechnungen
  - Wechsel zwischen Kernel-Threads aufwendiger

5 CPU-Scheduling

Der CPU-Scheduler entscheidet darüber, welcher Prozess als nächstes Rechenzeit auf der CPU erhält, indem er die Prozesse, welche Rechenzeit benötigen, in sog. Ready-Queues organisiert.

Man unterscheidet insbesondere zwischen *präemptiven* und *nicht präemptiven/kooperativen* Scheduling-Strategien: Bei präemptiven Strategien kann der Scheduler momentan auf der CPU bearbeitete Prozesse unterbrechen und erneut in die Ready-Queue einordnen. Bei nicht präemptiven Strategien muss der Prozess die Rechenzeit selber freigeben (möglich z.B. durch den Befehl `sched_yield()`) oder terminieren.

5.1 FIFO/FCFS

First In First Out/First Come First Serve. Nicht präemptiv.

Prozesse werden in der Reihenfolge abgearbeitet, in der sie beim CPU-Scheduler ankommen.

Vorteil: simple Implementation
Nachteil: Prozesse mit langer Laufzeit blockieren die CPU, kurze Prozesse werden benachteiligt.

5.2 LIFO/LCFS

Last In First Out/Last Come First Serve. Nicht präemptiv.

Es wird der Prozess bedient, der als letztes in den Zustand "ready" übergegangen ist

Vorteil: simple Implementation
Nachteil: Probleme wie bei FIFO

5.3 LIFO-PR

LIFO Preemptive Resume. Präemptive Variante von LIFO.

Gleiches Prinzip wie bei LIFO, nur können laufende Prozesse suspendiert werden, sobald ein neuer Prozess in den Zustand "ready" übergeht.

5.4 SPT/SJF

Shortest Processing Time/Shortest Job First. Nicht präemptiv.

Bedient stets den Prozess, der am schnellsten abzuarbeiten ist. Setzt voraus, dass die Bedienzeit der Prozesse vorhersehbar ist.

Vorteil: Realisiert die kürzestmögliche mittlere Wartezeit unter den nicht präemptiven Strategien.
Nachteil: Benachteiligt lange Prozesse. Praktisch nicht umsetzbar, da die Bedienzeit der Prozesse nicht exakt vorhergesehen werden kann.

5.5 SRPT

Shortest Remaining Processing Time. Präemptive Variante von SPT.

Gleiches Prinzip wie bei SPT. Kommt ein Prozess an, der eine niedrigere Bedienzeit besitzt als die verbleibende Bedienzeit des laufenden Prozesses, so wird dieser unterbrochen.

5.6 HRN

Highest Response Ratio Next. Nicht präemptiv.

Lange wartende Prozesse steigern nach und nach ihre Priorität. Dazu berechne für jeden Prozess die Response Ratio $r = \frac{\text{Wartezeit+Bedienzeit}}{\text{Bedienzeit}}$. Der Prozess mit größtem r wird als nächster bedient.

Vorteil: Verhindert, dass Prozesse lange Zeit nicht oder sogar nie bearbeitet werden.
Nachteil: Prozesse mit langer Laufzeit blockieren die CPU.

5.7 RR

Round Robin. Präemptiv.

Die Ready-Queue wird nach FIFO organisiert. Bekommt ein Prozess die CPU zugeteilt, so darf er diese jedoch nur für ein bestimmtes Quantum Q nutzen, bevor er unterbrochen und wieder hinten in die Ready-Queue eingereiht wird. FIFO ist ein Spezialfall von RR mit $Q = \infty$.

Vorteil: Fairste Aufteilung der Rechenzeit zwischen den Prozessen.
Nachteil: Bei zu kleinem Quantum geringe CPU-Auslastung durch zu häufige Kontextwechsel. Bei zu großem Quantum schlechte Antwortzeiten (tendiert zu FIFO).

5.8 EDF

Earliest Deadline First. Präemptiv.

Für diese sowie die folgende Strategie erhält jeder Prozess eine Deadline zugewiesen,

zu der der Prozess nach Möglichkeit abgearbeitet sein soll. EDF bearbeitet immer den Prozess mit der frühesten Deadline. Bei Ankunft eines neuen Prozesses mit früherer Deadline wird der aktuell bearbeitete Prozess suspendiert.

5.9 LLF

Least Laxity First. Präemptiv.

Diese Strategie betrachtet die Laxity, die "Entspanntheit", der wartenden Prozesse. Dies bezeichnet die Differenz zwischen Deadline und Restlaufzeit des Prozesses, also die Zeit, die der Prozess maximal warten kann um vor Ablauf der Deadline abgearbeitet zu sein. Es wird immer der Prozess als nächstes bearbeitet, bei dem diese Differenz am kleinsten ist (also der Prozess, welcher "am wenigsten entspannt"" ist).

6 Synchronisation

6.1 Anforderungen an wechselseitigen Ausschluss

Es müssen 3 Bedingungen erfüllt werden:

- Mutual Exclusion: Der wechselseitige Ausschluss selber
- Progress: Wenn mehrere Prozesse auf Eintritt in den kritischen Block warten, hängt die Wahl nur von den wartenden Prozessen selber ab, und wird in endlicher Zeit durchgeführt
- Bounded Waiting: Es gibt eine obere Grenze für Prozesse, die einen wartenden Prozess überholen können

6.2 Peterson-Algorithmus (für zwei Prozesse)

Folgendes Listing bezieht sich auf Prozess P_i:

```
while TRUE {
        flag[i] = TRUE;
        turn = j; //anderen Prozess Vorzug geben
        while (flag[j] && turn==j) { noop; } //anderer Prozess rechnet
        //kritischer Bereich;
        flag[i] = FALSE;
        //weitere Operationen;
}
```

6.3 Bakery-Algorithmus (für n-Prozesse)

1. Jeder Prozess zieht eine aufsteigende Nummer (kritisch!)

2. Für jeden Prozess j
   a) Gucke ob j gerade eine Nummer zieht, wenn ja warte bis j gezogen hat
   b) Überprüfe ob j's Nummer ($\neq 0$) kleiner als die des aktuellen Prozesses, wenn ja warte. Wenn die Nummern gleich sind (1. war kritisch) gebe dem Prozess mit der kleineren PID Vorrang
3. Führe den kritischen Bereich aus

6.4 Test-and-Set als atomare Operation

Auf den meisten Architekturen gibt es eine atomare Operation TSL (Test-and-Set), welche eine Variable auf `true` setzt und den alten Wert der Variablen zurückgibt. Damit lassen sich u.a. Semaphore einfach implementieren.

6.5 Semaphore/Mutexe

Die Synchronisierungsmechanismen von Semaphoren funktionieren über 3 grundlegende Funktionen:

- `init(s,n)`: initialisiert den Semaphor `s` mit dem Anfangswert `n`
- `wait(s)`: Wartet bis ein "Platz" in dem Semaphor `s` frei wird und nimmt diesen in Anspruch
- `signal(s)`: Gibt einen "Platz" frei

Mutexe sind Semaphore, mit Höchstwert 1. Um Bounded Waiting nicht zu verletzen ist es sinnvoll, bei der Implementation von Semaphoren wartende Prozesse in eine Warteschlange einzureihen.

6.6 (Bedingte) kritische Regionen

Eine kritische Region ist ein Bereich, in dem Variablen synchronisiert manipuliert werden. (Wie die Synchronisierung erfolgt ist implementationsabhängig). `region v do S` bedeutet: Führe `S` unter Synchronisation der Variablen in `v` aus.

Eine bedingte kritische Region ist `region v when w do S` und führt `S` aus, wenn zu den Einschränkungen einer normalen kritischen Region zusätzlich noch `w = True` gilt.

6.7 Monitore

Monitore bestehen aus einer Menge von Funktionen und Daten, wobei der Zugriff auf alle Bestandteile eines Monitors immer synchronisiert erfolgt. Der Monitor selbst hat hier die Aufgabe die Synchronisierung zu übernehmen. Dieses Konzept ist in Java z.B. mit dem Schlüsselwort `synchronized` realisiert.

6.8 Synchronisationsprobleme

6.8.1 Erzeuger/Verbraucher

Das Erzeuger/Verbraucher-Problem kann man lösen, indem man mit einem Mutex den beiden Parteien jeweils exklusiven Zugriff gewährt. Über einen weiteren Mutex lässt sich zudem signalisieren, ob ein Lager voll ist.

6.8.2 Reader/Writer

Beim Reader/Writer Problem dürfen Prozesse einer Klasse einen gleichzeitigen Zugriff auf einen Ressource bekommen. Das Problem wird gelöst, indem man einen Mutex für die Schreibphase und einen für die Lesephase einführt. Der Reader muss auch den Writer Mutex in Anspruch nehmen. Um die Freigabe der Mutexe zu lösen, muss man die Anzahl der aktiven Leser mitzählen.

6.8.3 Fünf Philosophen

Es gibt fünf Philosophen, welche an einem Tisch sitzen und denken. Zwischen ihnen liegen 5 Stäbchen. Die Philosophen essen und denken. Denken ist eine unkritische Phase, beim Essen hingegen wartet ein Philosoph bis das linke Stäbchen frei ist, nimmt es sich, und verfährt ebenso mit dem Rechten. Wenn jedoch alle Philosophen gleichzeitig essen, entsteht ein Deadlock.

6.8.4 3-Raucher-Problem

3 Raucher besitzen jeweils eins von 3 notwendigen Betriebsmitteln (Pfeife, Tabak, Feuerzeug). Eine unbekannte Person legt zwei zufällige Betriebsmittel hin. Der Raucher, dem die beiden Betriebsmittel fehlen nimmt sich diese, raucht, legt sie zurück, und der Agent tauscht die beiden von ihm ausgelegten Betriebsmittel gegen 2 zufällig gewählte Betriebsmittel.

7 Deadlocks

Die Deadlock-Vermeidung ist im Allgemeinen relativ schwierig und ressourcenverbrauchend.

7.1 Ressource-Allocation-Graph

Der Ressource-Allocation-Graph ist ein nützliches Tool zur Untersuchung von Deadlocks. Er besteht aus:

- Knoten: Prozesse (als Kreis dargestellt), Betriebsmittel (als Viereck dargestellt, gefüllt mit Punkten, welche für die Anzahl der Exemplare des Betriebsmittels stehen

- Kanten: Betriebsmittel → Prozess (wenn der Prozess das Betriebsmittel belegt), Prozess → Betriebsmittel (der Prozess wartet auf das Betriebsmittel)

7.2 Bedingungen für einen Deadlock

Damit ein Deadlock eintreten kann, müssen folgende Bedingungen erfüllt sein:

1. Circular-Wait: Kreis im Ressource-Allocation-Graph
2. Exclusive-Use: Keine gemeinsame Nutzung der Betriebsmittel möglich
3. Hold and Wait: Ein Prozess kann weitere Betriebsmittel anfordern, ohne welche zurückzugeben
4. Non-Preemption: Kein Entzung von Betriebsmitteln möglich

7.3 Banker-Algorithmus

Der Banker-Algorithmus wird zur Deadlock-Avoidance und Deadlock-Detection eingesetzt.

1. Für jeden nicht markierten Prozess P_i
   a) Prüfe ob es einen Prozess gibt, der insgesamt weniger Betriebsmittel in Anspruch nehmen muss, als zur Verfügung stehen. Wenn ja, erhöhe die zur Verfügung stehenden Betriebsmittel um die Menge, welcher P_i gerade hält, und markiere P_i (Wegen 1. kann P_i zu Ende laufen und alle seine Ressourcen frei geben

Wenn der Algorithmus für jede Betriebsmittelart immer alle Prozesse markiert, ist der Systemzustand sicher.

8 Speicherverwaltung

Bei der Speicherverwaltung unterscheidet man zwischen Paging und Segmentierung. Beim **Paging** wird der zu Verfügung stehende Speicher in kleine Rahmen aufgeteilt in die Seiten geladen werden können. Bei der **Segmentierung** werden die Daten, welche von einem Programm benötigt werden in große Segmente zusammengefasst und diese an eine freie Stelle im Hauptspeicher geladen. Um virtuelle Seiten- und Segmentadressen physischen Adressen zuzuordnen befindet sich in modernen CPUs eine *Memory Management Unit (MMU)*.

8.1 Segmentierungsstrategien

- First-Fit: Platziere das Segment in die erste passende Lücke
- Rotating-First-Fit: Wie First-Fit, suche jedoch ab der letzten Einfügeposition

- Best-Fit: Platziere das Segment in die kleinste passende Lücke
- Worst-Fit: Platziere das Segment in die größte passende Lücke

8.2 Buddy-Systeme bei Segmentierung

Das Buddy-System beruht auf einer Binärbaumstruktur und jede mögliche Segmentlänge ist eine 2er-Potenz. Die Wurzel wird so gewählt, dass der durch die Wurzel abgebildete Speicherbereich eine maximale Zweierpotenz bezüglich des verbauten Hauptspeichers ist.
Am Anfang eines jeden Buddy-Systems besteht der Baum nur aus der Wurzel. In jeder Ebene wird der Speicher aufgespalten und die Größe der einzelnen Knoten halbiert.
Je nach Anforderung an das System, wird der Baum so aufgeteilt, dass der Knoten mit dem kleinsten, aber ausreichend großen Speicherbereich, dem Segment zugeordnet wird. Defragmentierung lässt sich teilweise durch Vereinen leerer benachbarter Knoten erzielen.

8.2.1 Gewichtete Buddy-Syteme

Wenn in einem gewichteten Buddysystem ein Knoten der Größe 2^{r+2} gespalten wird, so haben die Kindknoten die Größen 2^r und $3 \cdot 2^r$. Dadurch lässt sich die Größe eines Segments feiner an die eventuellen Bedürfnisse anpassen.

8.3 Paging-Strategien

- FIFO (First In First Out): Bei Bedarf wird die älteste Seite im Hauptspeicher ersetzt.
- LRU (Least Recently Used): Ersetzt die Seite im Hauptspeicher, auf die am längsten nicht mehr zugegriffen wurde.
- Second Chance: Wird auf eine bereits im Hauptspeicher liegende Seite erneut zugegriffen, so wird das A-Bit auf 1 gesetzt. Im Bedarfsfall wird dann die älteste Seite mit nicht gesetztem A-Bit (A=0) verdrängt. Sind alle A-Bits gesetzt, so werden alle auf 0 zurückgesetzt.
- Second Chance - Clock: Die Seiten werden in einem Ringpuffer angeordnet. Bei Bedarf sucht ein Zeiger beginnend bei der ältesten Seite reihum nach einer Seite mit nicht gesetztem A-Bit. Währenddessen bewegt sich ein zweiter Zeiger mit festem Abstand zum ersten voraus und setzt jeweils das A-Bit an seiner aktuellen Position zurück. Dies begrenzt die Dauer der Suche nach einer Seite mit nicht gesetztem A-Bit.
- NRU (Not Recently Used): Erweiterung zu Second Chance: Zusätzlich zum A-Bit wird außerdem das D-Bit ("Dirty-Bit") nachgehalten. Dieses gibt an, ob die Seite verändert wurde (D=1) oder nicht (D=0), ob also schreibend auf die Seite zugegriffen wurde. So lässt sich eine Einteilung in vier Klassen vornehmen:

  - Klasse 1: A=0, D=0
  - Klasse 2: A=0, D=1
  - Klasse 3: A=1, D=0
  - Klasse 4: A=1, D=1

  Bei Bedarf wird die älteste Seite aus der niedrigsten nicht-leeren Klasse ersetzt.

- CLIMB: Die Seiten werden zunächst nach FIFO in einer Liste angeordnet (älteste Seite "unten"). Bei erneutem Zugriff auf eine Seite steigt diese um eine Position in der Liste nach oben. Bei Bedarf wird die unterste Seite verdrängt.

- Strategien mit Zugriffszähler: Diese Strategien nutzen für jede Seite einen Zähler, der auf verschiedene Weisen die Häufigkeit der Seitenzugriffe nachhält. Bei Bedarf wird die Seite mit dem kleinsten Zählerwert verdrängt.
  - LFU (Least Frequently Used): Bei jedem Zugriff wird der Zähler inkrementiert.
  - NFU (Not Frequently Used): Es werden periodisch die Zähler aller Seiten inkrementiert, auf die im vergangenen Zeitintervall zugegriffen wurde.
  - MFU (Most Frequently Used): Selbe Vorgehensweise wie LRU, jedoch wird bei Bedarf die Seite mit dem zweitniedrigsten Zählerwert verdrängt, da den niedrigsten Wert meist gerade erst geladene Seiten besitzen.

- OPT (Optimale Paging-Strategie): Verdrängt die Seite, die am längsten nicht mehr gebraucht werden wird. Dazu wird Wissen über zukünftige Seitenzugriffe benötigt, daher im allgemeinen Fall nicht verwendbar.

8.4 Lifetime-Funktion

Lifetime-Funktion $L(m) :=$ Anzahl der Seitenfehler pro Zugriffe (gemittelt) bei m zur Verfügung gestellten Rahmen.
Das *ominöse-Knie-Kriterium*: Zeichnet man den Graphen der Lifetime-Funktion und legt eine Tangente beginnend bei dem Punkt $(0, 1)$ an die Lifetime-Funktion an. Dieser Schnittpunkt ist die optimale Rahmenzahl für diesen Prozess.

9 Dateien

Wichtig: Bevor auf ein Dateisystem zugegriffen werden kann, muss es im System in den aktuellen Verzeichnisbaum eingegliedert werden (*Mounten*).

9.1 Zugriffsrechte

Die Zugriffsrechte werden in $3 \cdot 3 = 9$ Bits gespeichert. Die ersten 3 Bits, geben die Zugriffsrechte des Eigentümers an, die zweiten 3 Bits die Zugriffsrechte der Gruppe der Datei und die letzten 3 Bits die Zugriffsrechte für alle anderen. Die 3 Bits stehen jeweils

für Schreibzugriff, Lesezugriff, Ausführungsrecht. Der Einfachheit halber werden diese im Oktalsystem angegeben. Beispiele:

- $760_8 = 111|110|000_2$ - der zugehörige Benutzer darf die Datei lesen, schreiben und ausführen, die zugeordnete Gruppe darf die Datei lesen und schreiben, alle anderen dürfen nichts.
- $755_8 = 111|101|101_2$ - der zugehörige Benutzer darf alles, alle anderen dürfen die Datei lesen und ausführen

9.2 Links

Man unterscheidet unter Linux zwei Arten von Links:

1. Symbolische (`$ ln -s Ziel Quelle`): In einem symbolischen Link ist der Pfad der Datei hinterlegt, auf welche verwiesen wird.
2. Hardlinks (`$ ln Ziel Quelle`): Hardlinks zeigen hingegen direkt auf den I-Node. Damit funktioniert der Link auch noch nach z.B. einem Umbenennen der eigentlichen Datei.

9.3 File Descriptor Table

Beim Öffnen einer Datei wird unter Linux-System immer ein neuer File Descriptor angelegt. Die Verwaltung dieser File Deskriptoren (und damit die Verwaltung der geöffneten Dateien) erfolgt in der File Descriptor Table. In dieser Tabelle stehen für jeden File Descriptor u.a. Zugriffsrechte, Eigentümer, aktuelle Position des "Cursors" innerhalb der Datei .... Wenn man z.B. eine Datei mittels **read** liest, wird immer von der letzten Position die man gelesen hat weitergelesen.

10 Dateisystemaufbau (ext2)

10.1 I-Nodes

I-Nodes (Index-Nodes), im Allgemeinen auch FCB (File Control Block) genannt, dienen zur Repräsentation von Datein und Verzeichnissen und speichern wichtige Eigenschaften einer Datei(/Verzeichnis) (z.B. Typ, Zugriffsrechte Größe, letzte Änderung, ...). Auf die Datei wird wie folgt referenziert:

1. direct: 12 Pointer auf Blöcke
2. single indirect: Adresse eines Blockes der die Adressen der DateiBlöcke beeinhaltet
3. double indirect: Adresse eines Blockes der die Adressen weiterer Blöcke beeinhaltet, die die Adresser der finalen Dateiblöcke beeinhalten
4. triple indirect: Eine Indirektionsstufe mehr als double indirect

10.2 Blockgruppen

Die Festplatte wird in sogenannte Blockgruppen unterteilt. Die ersten 512 Bytes der Festplatte enthalten den MBR (Master Boot Record) (zum Laden des Bootloaders), und die erste Blockgruppe den Bootloader. Jede andere Blockgruppe ist folgendermaßen unterteilt:

1. Superblock: Enthält Informationen über das Dateisystem (Bspsw.: Root-Inode, wie viele I-Nodes und Datenblöcke die Gruppe enthält, etc.)
2. Group Descriptor: Informationen über die Position Block-Bitmap, Anzahl freier I-Nodes und Informationen der anderen Blockgruppen (Redundanz)
3. Block-Bitmap/I-Node Bitmap: Kennzeichnen welche Blocks/I-Nodes frei/belegt sind
4. I-Nodes
5. Data Blocks

10.3 Journaling

Operationen auf Dateisystemen können verzögert stattfinden (Effizienzgründe) und im Falle eines Absturzes zu *Inkonsistenz* führen. Lösung: Führe ein *Journal* (Tagebuch) ein, indem alle Operationen dokumentiert werden. Ausgeführte Einträge werden wieder gelöscht, im Falle eines Absturzes reicht es also das Journal durchzugehen, um die Konsistenz wiederherzustellen.

11 I/O

11.1 I/O-Hardware

Zur Steuerung des I/O-Systems enthalten Rechner häufig:

- Controller: Chip auf Motherboard oder (PCI-)Karte, welcher Anschlüsse für I/O-Geräte bereitstellt und diesen Gerätetyp steuert (etwa USB-Controller). Ein Controller kann zudem über einen *Buffer* verfügen (Performanz)
- I/O-Werk: Dient zur Steuerung der I/O-Geräte und sitzt zwischen CPU und Controller. Vorteil: Da das I/O-Werk die Steuerung übernehmen kann, spart man CPU-Zeit

11.2 I/O-Controller

Die Kommunikation mit dem Controller erfolgt über spezielle Register. Man unterscheidet zwischen *Kontrollregistern* (Status und Control) und *Datenregister* (Data-In und Data-Out).

11.3 I/O-Werk

Einfachster Fall: CPU simuliert das I/O-Werk. Dies nennt sich PIO (Programmed Input Output)

Moderne autonome I/O-Werke verwenden meistens das Prinzip *DMA* (Direct Memory Access) zur Kommunikation mit der CPU. Das I/O-Werk kann so Daten zwischen Controller und Hauptspeicher hin- und herbewegen. Die I/O-Geräte selbst können über Register addressiert werden oder in dem man ihnen eine Adresse im Hauptspeicher zuordnet (Memory-mapped I/O).

12 Disk-Scheduling

- FCFS (First Come First Serve): Zugriffsoperationen werden der Reihe nach abgearbeitet
- SSTF (Shortest Seek Time First): Als nächstes die Zugriffsoperation bedienen, die am nächsten am aktuellen Zylinder liegt
- SCAN: Bewege den Zylinderkopf von links nach rechts und zurück und arbeite dabei die Zugriffsoperationen ab
- LOOK: Wie Scan nur ändere die Richtung schon bei der äußersten Zugriffsoperation
- C-SCAN: In einer Richtung Abarbeitung der Zugriffsoperationen, in der Rückrichtung nur bewegen (Definition C-Look analog)
- Noop: Wie *FCFS*, nur wenn bei Zylinder i auch eine Zugriffsanfrage auf Zylinder $i+1$ vorliegt, führe diese Anfrage als nächstes aus
- Deadline: Führe für eine festgelegte Zeitdauer *C-Look* aus und bediene dann alle Anfragen mit abgelaufener Deadline. Repeat.
- Anticipatory: C-Look jedoch nach jeder ausgeführten Operation kurzzeitig SSTF ausführen
- Completely Fair Queuing: *Antipatory-Scheduling* mit unterschiedlich priorisierten Warteschlangen

13 Anhang

13.1 Shell Befehlstabelle

Befehl	*Erklärung*
`wc`	Word Count
`echo`	Gibt die Argumente aus
`awk`	Praktisches Tool das als Argument eine Zeichenkette einer eigenen Turingvollständigen Sprache übergeben bekommt und diese auswertet. Beispiel: `ls -l | awk '{print $3}'`
`stat`	Informationen zu Dateien anzeigen (funktioniert über *I-Nodes*)

13.2 Syscall-Tabelle

Befehl	*Erklärung*
`pid_t fork(void)`	Klont den aktuellen Prozess. Der Vaterprozess erhält die PID des Kindes als Rückgabewert und das Kind 0.
`int pipe(int fd[2])`	Erstellt eine Pipe mit Leseeingang `fd[0]` und Schreibeingang `fd[1]`.
`int dup2(int oldfd, int newfd)`	Stellt eine Verbindung von dem Filedeskriptor `oldfd` zum Fildeskriptor `newfd` her.