Rolf Klein, Christian Icking, Lihong Ma

Betriebssysteme und Rechnernetze

Kurseinheit 2:
Hauptspeicher und Dateisysteme

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Der Inhalt dieses Studienbriefs wird gedruckt auf Recyclingpapier (80 g/m^2, weiß), hergestellt aus 100 % Altpapier.

Inhalt

Teil I: Betriebssysteme

Teil II: Rechnernetze

Die Autoren

Univ.-Prof. Dr. rer. nat. Rolf Klein

Studium der Mathematik und mathematischen Logik an der Universität Münster zum Dipl.-Math. 1978, dann Universität Erlangen-Nürnberg, hier Promotion in Mathematik 1982, danach Universität Karlsruhe, University of Waterloo in Kanada, Universität Freiburg, dort Habilitation in Informatik 1989. Hochschulprofessor an der Universität Essen, der FernUniversität in Hagen, seit 2000 an der Universität Bonn, seit 2019 emeritiert.

apl. Prof. Dr. rer. nat. Christian Icking

Studium der Mathematik und Informatik an der Universität Münster und der Université Pierre und Marie Curie Paris VI zur Maîtrise in Mathematik 1984, dann auch École Nationale Supérieure de Techniques Avançées Paris zum DEA (Diplom) Informatik 1985, danach Universität Karlsruhe, Universität Freiburg, Universität Essen und FernUniversität in Hagen, hier Promotion und Habilitation in Informatik 1994 und 2002, außerplanmäßiger Professor 2010.

Dr. rer. nat. Lihong Ma

Studium der Mathematik an der Universität Yunnan, Bachelor in Mathematik 1984, dann Technische Universität Kunming, Universität Karlsruhe, Universität Freiburg, dort Dipl.-Math. 1990, danach Universität Essen und FernUniversität in Hagen, hier Promotion in Informatik 2000, wissenschaftliche Mitarbeiterin.

Kurseinheit 2

Hauptspeicher und Dateisysteme

In der ersten Kurseinheit haben wir uns einen Überblick über verschiedene konkrete Aufgaben eines Betriebssystems verschafft. Jetzt wollen wir auf drei wichtige Bereiche näher eingehen, mit denen Anwendungs- und Systemprogrammierer in Berührung kommen: Hauptspeicherverwaltung, Synchronisation und Dateisysteme.

2.1 Hauptspeicherverwaltung

In modernen Betriebssystemen sind Prozesse die elementaren Arbeitseinheiten. Eine der Hauptaufgaben eines Betriebssystems besteht darin, die existierenden Prozesse quasi-parallel auf der CPU ablaufen zu lassen. Dabei sollte es fair zugehen, die Prozesse sollen sich gegenseitig nicht behindern, und jeder Prozess soll möglichst schnell bearbeitet werden.

Damit der Prozesswechsel ohne Schwierigkeiten funktioniert, werden für jeden bereiten oder blockierten Prozess die Inhalte des Befehlszählers und der übrigen Register zum Zeitpunkt der letzten Unterbrechung im Prozesskontrollblock gespeichert; vergleiche Abschnitt 1.4.1. Man muss aber auch sicherstellen, dass die Inhalte der Hauptspeicherbereiche erhalten bleiben, die den existierenden Prozessen zugewiesen sind. Hier stellt sich die Frage, wie diese Speicherplatzzuweisung überhaupt erfolgt. Darauf wollen wir jetzt eingehen.

In Abschnitt 1.3.6 sind wir davon ausgegangen, dass das Betriebssystem jedem Prozess einen zusammenhängenden Hauptspeicherbereich zuweist, der als *physischer Adressraum* bezeichnet wird. Eine *physische Adresse* zeigt auf eine aktuelle Speicherstelle im Hauptspeicher. Der Prozess selbst braucht die physischen Hardwareadressen nicht zu kennen; er kann *logische Adressen* verwenden. Eine logische Adresse ist ein Verweis auf eine Speicherstelle, die unabhängig vom physischen Hauptspeicher ist, zum Beispiel die relative Position eines Befehls in einem Programmstück. Ein Programm erzeugt eine Menge von logischen Adressen, die zusammen den *logischen Adressraum* des Prozesses bilden. Bevor ein Zugriff auf den Hauptspeicher stattfindet, muss eine logische Adresse auf eine physische abgebildet werden. Die Abbildung der logischen auf die physischen Adressen ist die Aufgabe des Betriebssystems. Sie wird oft mit der Hilfe einer Hardware, der *MMU* (Memory Management Unit), die eine logische Adresse von der CPU erhält und eine physische Adresse ausgibt, erledigt.

physische Adresse

logische Adressen

logischen Adressraum

MMU

2.1.1 Zusammenhängende Hauptspeicherzuweisungen

Jeder Prozess soll ein Stück vom Hauptspeicher zugewiesen bekommen. Die einfachste Zuweisungsstrategie ist, dass jeder einen zusammenhängenden Bereich im Hauptspeicher erhält. Es gibt zwei Möglichkeiten:

- Der Hauptspeicher wird fest in zusammenhängende Bereiche unterschiedlicher Größe aufgeteilt, und jeder Prozess bekommt einen davon, dessen Größe er mindestens benötigt. Der vom Prozess nicht benutzte Teil des Bereiches ist die sogenannte *interne Fragmentierung*.

interne Fragmentierung

- Jeder Prozess bekommt einen zusammenhängenden Bereich genau der Größe, die er auch angefordert hat.

In beiden Fällen ist die Abbildung einer logischen Adresse auf die physische sehr einfach. Dafür brauchen wir zwei speziellen CPU-Register:

Basisregister

- *Basisregister*: Zur Speicherung der Startadresse eines Adressraums.

Grenzregister

- *Grenzregister*: Zur Speicherung der Länge des logischen Adressraums.

Die physische Adresse zu einer logischen Adresse ergibt sich durch Addition der Startadresse. Wir sprechen hier auch von *relativer Adressierung* (relativer zum Basis-Register). Abbildung 2.1 zeigt ein Beispiel zur Umrechnung der Adressen.

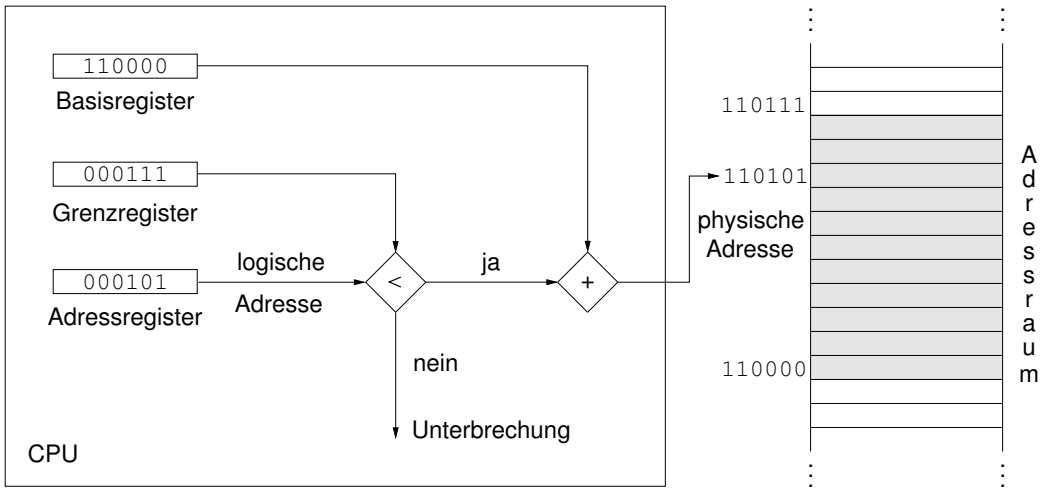

Abbildung 2.1: Die Abbildung logischer Adressen auf physische Adressen bei zusammenhängenden Adressräumen.

Vor der Addition testet die MMU, ob die logische Adresse kleiner als die im Grenzregister ist. Falls nein, erzeugt die MMU eine Unterbrechung, so wird der Speicherschutz gewährleistet. Durch die beiden Register wird gleichzeitig erreicht, dass die Abbildung auf die physischen Adressen flexibel ist, das heißt, dass Programme *relokierbar* bleiben. Sie können zur Laufzeit in verschiedene Bereiche des Hauptspeichers geladen werden.

Wenn viele Prozesse gleichzeitig existieren, kann es vorkommen, dass der Hauptspeicher zu klein wird, um alle physischen Adressräume aufzunehmen. In

diesem Fall werden einige bereite oder blockierte Prozesse durch ein Verschieben ihres Adressraums in den Sekundärspeicher ausgelagert. Die Entscheidung darüber, welche Prozesse ausgelagert werden, trifft ein *Langzeit-Scheduler*.[1]

Langzeit-Scheduler

Bevor ein ausgelagerter Prozess weiter rechnen kann, muss er erst wieder eingelagert werden; der Vorgang des Aus- und Einlagerns wird im Englischen als *swapping* bezeichnet. Beim Einlagern ist es dank der Relokierbarkeit nicht notwendig, den Prozess an exakt dieselbe Stelle im Hauptspeicher zurückzuschreiben, an der er vorher gestanden hat.

swapping

Weil sich die Menge der existierenden Prozesse ständig ändert, entstehen zwischen den physischen Adressräumen der Prozesse im Hauptspeicher zwangsläufig Lücken; dieses Phänomen nennt man *externe Fragmentierung*. Es kann sein, dass keine der vorhandenen Lücken ausreicht, um den physischen Adressraum eines neu erzeugten Prozesses aufzunehmen. Dann müsste ein Prozess ausgelagert werden, obwohl insgesamt noch ausreichend Hauptspeicher frei ist – ein wenig effizientes Vorgehen.

externe Fragmentierung

Zur Vermeidung dieses Problems kann man verschiedene Strategien anwenden. Eine Möglichkeit besteht darin, die im Hauptspeicher eingelagerten Prozesse hin und wieder zusammenzuschieben, damit aus vielen kleinen Lücken eine große wird. Eine solche *Kompaktifizierung* ist aber mit einem hohen Aufwand verbunden.

Kompaktifizierung

Man gewinnt mehr Flexibilität in der Speichervergabe, wenn man die Forderung aufgibt, dass jeder Prozess ein *zusammenhängendes* Stück vom Hauptspeicher bekommen soll. Stattdessen kann man zum Beispiel jedem Prozess mehrere zusammenhängende *Segmente* im Hauptspeicher zuweisen, die unterschiedlich lang sein dürfen. Dieser im Englischen als *segmentation* bezeichnete Ansatz ist auch aus der Sicht des Programmierers sinnvoll: Man kann verschiedene Programmodule in unterschiedlichen Segmenten unterbringen und zum Beispiel ein weiteres Segment für die Daten verwenden. Die Adressierung innerhalb eines Segments erfolgt analog zu Abbildung 2.1, zur Bezeichnung des Segments wird zusätzlich eine Segmentnummer angegeben.

segmentation

2.1.2 Paging

Ganz beseitigt wird das Problem der externen Fragmentierung, wenn man den Hauptspeicher in sehr viele kleine Stücke gleicher Größe aufteilt und jedem Prozess die erforderliche Anzahl solcher Stücke zuweist; diese Stücke brauchen dabei im Hauptspeicher nicht hintereinander zu liegen. Dieser Ansatz heißt auf Englisch *paging*. Man teilt den logischen Speicher[2] in gleichgroße Stücke auf, die *Seiten* (pages) genannt werden. Der physische Speicher wird in *Seitenrahmen* (frames)[3] aufgeteilt. Eine Seite passt genau in einen Seitenrahmen. Die *Seitentabelle* (page table) legt fest, welche Seite in welchem Seitenrahmen steht,

paging

Seitentabelle

[1] Der Adressraum eines ausgelagerten Prozesses steht für das Betriebssystem nicht zur Verfügung – ein weiterer Grund dafür, dass nach Ausführung eines Leseauftrags die gelesenen Daten zunächst im systemeigenen Speicher abgelegt werden; vergleiche Abschnitt 1.5.

[2] Als logischen Speicher bezeichnen wir die Organisation des Adressraums eines Prozesses aus Benutzersicht/Programmiererersicht.

[3] Rahmen (frames) werden gelegentlich auch Kacheln genannt.

und liefert damit eine Abbildung vom logischen auf den physischen Speicher; siehe Abbildung 2.2.

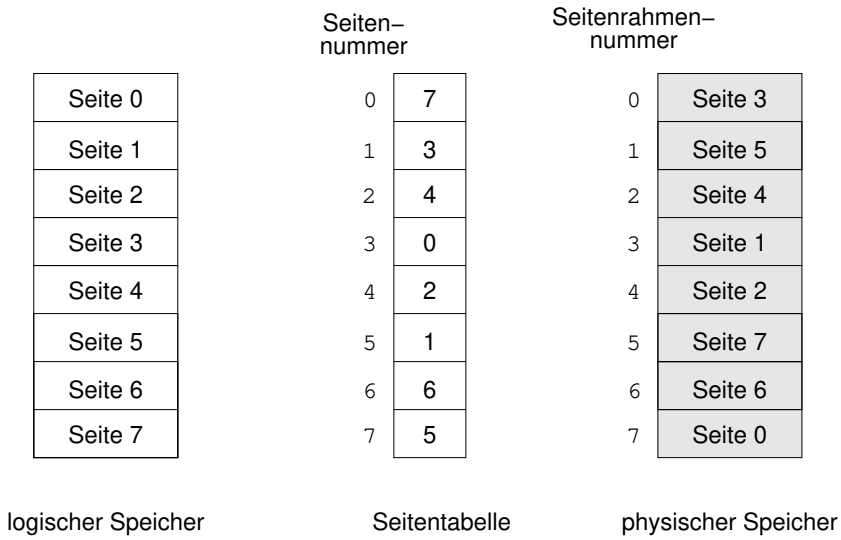

Abbildung 2.2: Die Seitentabelle bildet Seiten auf Seitenrahmen ab.

Wie funktioniert aber bei Paging die Umrechnung einer logischen Adresse in die physische?

Als Beispiel betrachten wir einen logischen Adressraum der Größe $2^{10} = 1024$ Byte mit einer Seitengröße 128 Byte, siehe Abbildung 2.3. Hier zeigt jede logische Adresse auf ein Byte. Also muss es insgesamt 1024 Adressen geben, von 0 bis 1023. Um die 1024 Adressen darzustellen, benötigen wir 10 Bit. Also ist jede Adresse des logischen Adressraums eine binäre Zahl der Länge 10 Bit. Legen wir nun die Seitengröße mit 128 Byte fest, dann hat der logische Adressraum in unserem kleinen Beispiel insgesamt

$$\frac{1024 \text{ Byte}}{128 \text{ Byte}} = \frac{2^{10}}{2^7} = 2^3 = 8$$

Seiten. Man kann nun die Anzahl der logischen Adressen

$$1024 = 8 \cdot 128 \quad \text{oder} \quad 2^{10} = 2^3 \cdot 2^7$$

auch als Produkt der Anzahl der Seiten und der Seitengröße schreiben. Daraus ergibt sich die die Idee der für das Paging typischen Aufteilung von logischen Adressen in zwei Abschnitte: Der vordere Teil, hier die ersten 3 Bit, stellt die *Seitennummer* und der hintere Teil, hier die letzten 7 Bit, den *Offset* dar.

Seitennummer
Offset

Rechnen wir nun konkrete Beispiele durch. Wenn wir wissen wollen, in welcher Seite z. B. die logische Adresse 254 liegt, dann berechnen wir die Division mit Rest von 254 durch 128 und erhalten den Quotient 1 mit Rest 126, da

$$254 = 1 \cdot 128 + 126$$

gilt. Dies bedeutet, dass die logische Adresse 254 in der Seite mit Nummer 1 liegt und innerhalb der Seite die relative Position 126 hat.

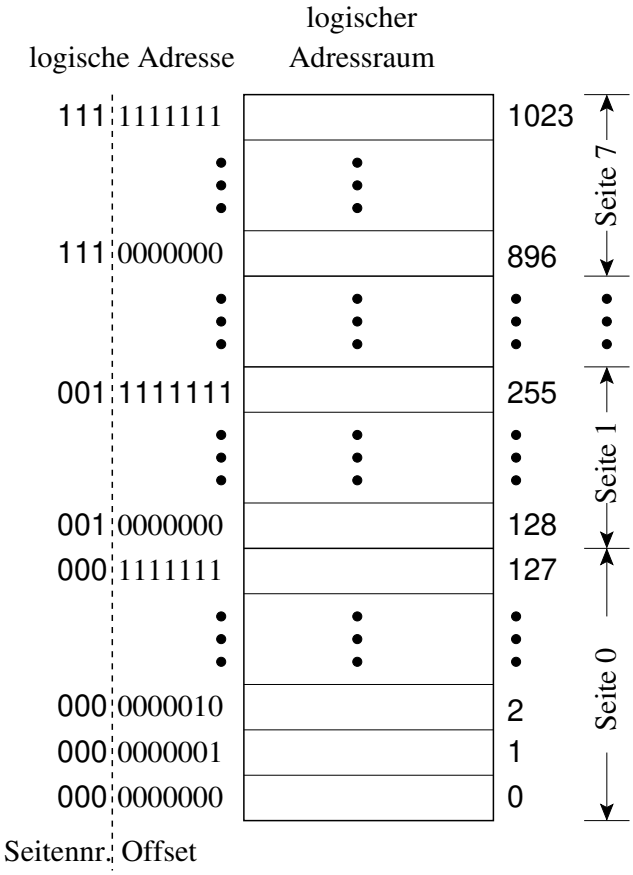

Abbildung 2.3: Ein logischer Adressraum wird in 8 Seiten der Größe 128 Byte aufgeteilt. Die ersten drei Bits einer Adresse stellen die Seitennummer der Adresse dar. Die restlichen 7 Bits sind der Offset, die Position der Adresse innerhalb der Seite.

Übungsaufgabe 2.1 Ein Prozess benötigt zur Laufzeit einen logischen Speicherbereich von 8000 Bytes. Wieviele Seiten muss das Betriebssystem diesem Prozess zuteilen, wenn die Seitenrahmengröße 1024 Byte beträgt? Wie groß ist die interne Fragmentierung bei dieser Zuweisung?

`https://e.feu.de/1801-seitenzahl`

Die MMU hat es aber noch einfacher als wir, die lieber im Dezimalsystem rechnen. Sie bekommt die binäre Zahl 0011111110 und interpretiert die ersten drei Bits 001 als die Seitennummer 1 und die letzten 7 Bits 1111110 als den Offset 126, die Division mit Rest ergibt sich mühelos durch die Aufteilung der Bits an der richtigen Stelle. Da eine Seite genau so groß wie ein Seitenrahmen ist, bleibt der Offset einer logischen Adresse gleich dem physischen Offset im Seitenrahmen. Die Konsequenz ist, dass ein Eintrag in der Seitentabelle einfach nur die Nummer des Seitenrahmens zu sein braucht, in dem die Seite auch steht.

Wir fassen die Erkenntnisse aus diesem Beispiel zusammen:

- Die Bits einer logischen Adresse setzen sich aus Seitennummer und Offset zusammen.
- Der Offset ist die Position des Bytes innerhalb einer Seite und eines Seitenrahmens.
- Der Offset in einer logischen Adresse ist derselbe wie in der zugehörigen physischen Adresse.
- Ein Index in die Seitentabelle ist eine Seitennummer und ein Eintrag in der Seitentabelle ist eine Seitenrahmennummer.
- Die Bits einer physischen Adresse werden aus Seitenrahmennummer und Offset zusammengesetzt.
- Für die Umrechnung von logischen in physische Adressen braucht die MMU nur die ersten Bits der Adresse als Seitennummer zu entnehmen und sie als Index in der Seitentabelle zu verwenden. Dort steht die zugehörige Seitenrahmennummer, die nur noch mit dem Offset zusammengesetzt werden muss, um die physische Adresse zu erhalten.

Wir betrachten als Beispiel, wie die MMU die logische Adresse

$$254 = 0011111110$$

auf die physische Adresse abbildet, siehe dazu Abbildung 2.4.

Die MMU erhält die Binärzahl 0011111110, entnimmt die ersten drei Bits 001, sucht in der Seitentabelle an der Stelle 001 nach der Seitenrahmennummer und erhält die Zahl 011. Nun setzt die MMU sie mit dem Offset 1111110 zusammen und erhält die physische Adresse 0111111110, das ist umgerechnet

$$0111111110 = 3 \cdot 128 + 126 = 510.$$

Das Betriebssystem legt die maximale Größe eines logischen Adressraums fest, unabhängig von einzelnen Prozessen. Es legt auch die Seitengröße fest und damit auch, wie viele Seiten der logische Adressraum eines Prozesses maximal haben kann. Dadurch liegt auch die Anzahl der Bits für die Seitennummer und für den Offset fest. Nun konfiguriert das Betriebssystem die MMU so, dass sie immer die gleiche Anzahl von ersten Bits von einer logischen Adresse als Seitennummer entnimmt und sie als Index für die Suche nach der Seitenrahmennummer in der Seitentabelle verwendet.

Die Seitentabelle ist Teil des Prozesskontextes. Weil jeder Hauptspeicherzugriff zunächst einen Zugriff auf die Seitentabelle auslöst, kommt es hier auf Effizienz an. Deshalb werden einige Einträge einer Seitentabelle, die momentan verwendet werden, oft in einem schnellen Speicher auf dem CPU-Chip bzw. in der MMU gehalten.

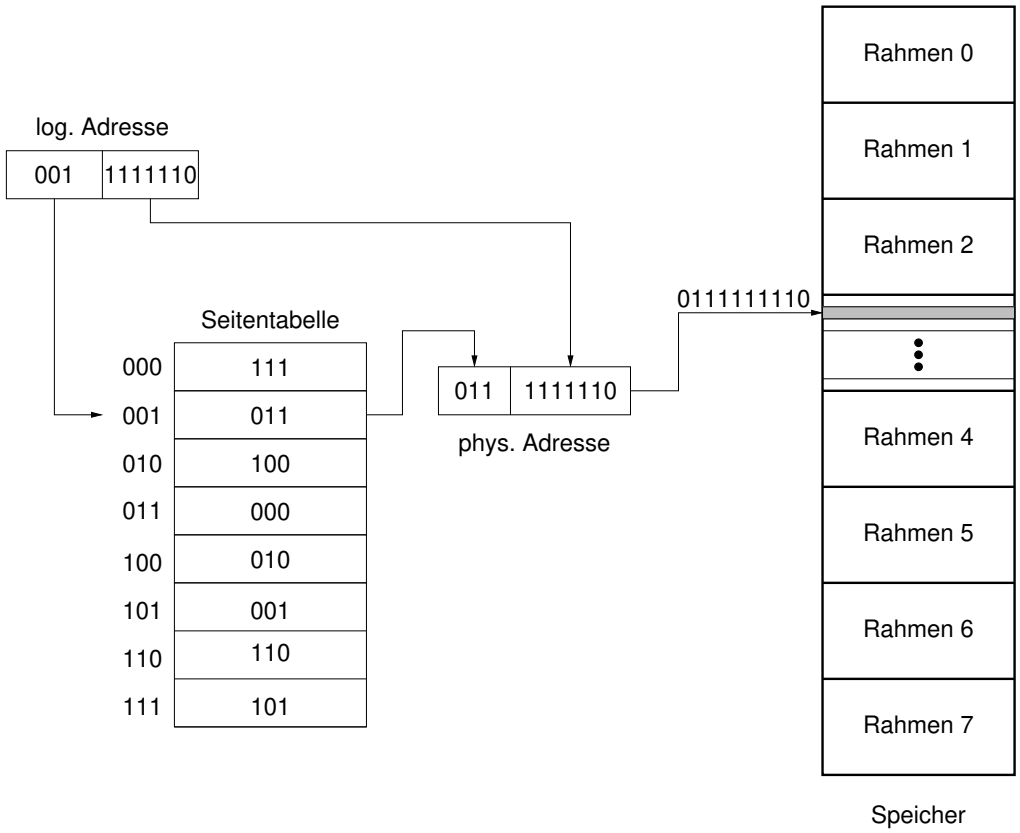

Abbildung 2.4: Adressberechnung beim Paging.

Übungsaufgabe 2.2 Warum darf ein Prozess nicht auf seine Seitentabelle zugreifen?

`https://e.feu.de/1801-seitentabelle-prozesskontext`

Durch die Verwendung von Seiten kann zwar keine externe Fragmentierung auftreten; weil aber immer nur ganze Seiten zugeteilt werden, lässt sich eine gewisse interne Fragmentierung nicht vermeiden. Zum Beispiel könnte ein Prozess eigentlich nur 3,1 Seiten benötigen, und von den vier zugeteilten Seiten würde die letzte zu 90 % ungenutzt bleiben. Dies stellt aber kein Problem dar bei aktuellen Systemen mit Hunderten von Prozessen und Millionen von Seiten.

externe Fragmentierung

interne Fragmentierung

Übungsaufgabe 2.3 Welche Auswirkungen hat es, wenn man die Seiten sehr groß oder sehr klein auslegt?

`https://e.feu.de/1801-seitengroesse`

Übungsaufgabe 2.4 Da die Seitentabellen oft recht groß werden, schnelle Hardwareregister jedoch teuer sind, hält man oft die gesamte Seitentabelle im Hauptspeicher und sieht im Prozessor einen Registersatz als Cache-Speicher für einige Einträge der Seitentabelle vor. Angenommen, die Zugriffszeiten für einen solchen Cache beträgt 20 ns und für den Hauptspeicher 100 ns. Um wieviel Prozent steigt die durchschnittliche Zeit für einen lesenden und schreibenden Zugriff gegenüber einem System ohne Paging, wenn die Trefferrate für den Cache 80 % (98 %) beträgt?

`https://e.feu.de/1801-cache-seitentabelle`

Zuweisungstrategie

Um Prozessen große Hauptspeicherbereiche zuweisen zu können, braucht das Betriebssystem effiziente Mechanismen zur Verwaltung der belegten und freien Seiten, dabei kann auch eine höhere interne Fragmentierung in Kauf genommen werden. Als Beispiel für eine Zuweisungstrategie betrachten wir ein Verfahren in Linux:

Beispiel Linux

Buddy-Strategie

Für den physischen Speicher gibt es in Linux einen *Seitenallokierer*, der Bereiche von aufeinander folgenden Seitenrahmen bereitstellen kann. Dabei bedient er sich der sogenannten *Buddy-Strategie*: Der Hauptspeicher besteht aus zusammenhängenden Stücken, die jeweils eine Zweierpotenz viele Seiten enthalten. Ein Stück ist entweder belegt oder frei. Wenn der Allokierer einen zusammenhängenden Bereich einer bestimmten Länge benötigt, nimmt er das kleinste freie Stück, das mindestens die erforderliche Länge aufweist. Wenn es mehr als doppelt so lang ist wie der benötigte Bereich, so wird es halbiert; das eine halbe Stück wird benutzt, das andere – sein *Buddy*[4] – bleibt frei. Wann immer ein Stück frei wird und sein Buddy bereits frei ist, werden sie wieder verschmolzen. Dieses Verfahren ist recht einfach zu implementieren, kann aber zu erhöhter interner Fragmentierung führen, denn wenn ein Prozess 33 Seiten benötigt, belegt er schon 64.

gemeinsame Speicherbereiche

Bei Verwendung von mehreren Segmenten oder Seiten je Prozess kann man neben den "privaten" Speicherbereichen, auf die nur der Prozess selbst zugreifen darf, auch "öffentliche" Bereiche einrichten, die von mehreren Prozessen gemeinsam benutzt werden können: Es genügt ja, bei jedem Segment oder jeder Seite zu vermerken, welche Prozesse Schreib- oder Leserecht daran haben. So kann zum Beispiel das Code-Segment eines Pascal-Compilers von mehreren Anwendern gleichzeitig benutzt werden, ohne dass jeder Prozess eine eigene Kopie des Compilerprogramms benötigt. Natürlich benötigt jeder Prozess sein eigenes Datensegment und Stacksegment. Außerdem kann man gemeinsame Speicherbereiche für die Kommunikation zwischen Prozessen benutzen; vergleiche den Schluss von Abschnitt 1.6. Hierauf gehen wir in Abschnitt 2.3 näher ein.

[4] Im Englischen bedeutet *buddy* soviel wie Kamerad, Kumpan.

2.1.3 Virtueller Hauptspeicher

virtueller Speicher

Am Ende dieses Abschnitts wollen wir die Technik des *virtuellen Speichers* (virtual memory) zur Hauptspeicherverwaltung besprechen, die heute große Bedeutung erlangt hat und das in praktisch allen modernen Betriebssystemen implementiert ist. Es kombiniert zwei Ansätze, die wir oben vorgestellt haben: die Einteilung des physischen Speichers in Seitenrahmen und die Idee, nur die Informationen im Hauptspeicher zu halten, die gerade benötigt werden.

Für die Programmierer ist die Verwendung virtuellen Speichers sehr angenehm: Sie können große logische Adressräume verwenden, ohne auf mögliche physische Grenzen achten zu müssen.

Übungsaufgabe 2.5 Wodurch ist die maximale Größe des logischen Adressraums begrenzt?

`https://e.feu.de/1801-logischer-adressraum`

Ein bereiter Prozess wird rechnend gemacht, auch wenn nicht alle seine Seiten in Seitenrahmen des Hauptspeichers stehen; die fehlenden Seiten stehen im Sekundärspeicher und sind in der Seitentabelle entsprechend markiert. Das Betriebssystem merkt sich, wo die fehlenden Seiten im Sekundärspeicher stehen. Ein Zugriff auf eine Seite, die nicht im Hauptspeicher steht, wird als *Seitenfehler* bezeichnet.

Seitenfehler

Wie kann die MMU wissen, ob eine Seite im Hauptspeicher vorhanden ist, wenn sie in der Seitentabelle nach der Seitenrahmennummer sucht? Dazu wird in der Seitentabelle für jeden Eintrag zusätzlich ein *present-Bit* verwaltet, das zeigt, ob die Seite im Hauptspeicher vorliegt. Im Fall, dass die Seite nicht im Hauptspeicher vorhanden ist, löst die MMU eine Software-Unterbrechung (trap) aus, da die Software auf eine derzeit nicht verfügbare Seite zugreifen will; vergleiche Abschnitt 1.3.4. Die fehlende Seite wird von der Festplatte gelesen, danach kann der Prozess weiterrechnen. Diese Technik wird als *demand paging* bezeichnet, bei der eine Seite erst in den Hauptspeicher eingelagert wird, wenn sie auch gebraucht wird.

present-Bit

demand paging

Es kann beim Einlagern einer Seite vorkommen, dass kein Seitenrahmen mehr frei ist. Dann muss eine andere Seite in den Sekundärspeicher ausgelagert werden. Für die Wahl der auszulagernden Seite gibt es verschiedene Strategien; hier liegt dieselbe Situation vor wie beim Hauptspeicher-Cache, den wir in Abschnitt 1.2.2 betrachtet hatten.

- Die *optimale Strategie* lagert diejenige Seite aus, die erst am weitesten in der Zukunft wieder benötigt wird, um künftige Seitenfehler möglichst zu vermeiden. Leider steht diese Information in der Regel nicht zur Verfügung.[5] Man muss sich daher mit sub-optimalen Strategien begnügen.

optimale Strategie

[5] Wenn k Seitenrahmen vorhanden sind, können dabei höchstens k mal so viele Seitenfehler entstehen wie bei optimaler Auslagerung, und in Unkenntnis der Zukunft lässt sich auch kein besseres Ergebnis erreichen. Wer sich für solche *on-line-Strategien* interessiert, sei auf Fiat und Woeginger (Hrsg.) [1] verwiesen.

LRU

- Die Strategie *LRU* (least recently used) lagert die Seite aus, deren letzte Benutzung am weitesten zurückliegt. LRU ist eine Annäherung der optimalen Strategie.

dirty-Bit

- Es soll möglichst eine Seite ausgelagert werden, die seit der letzten Einlagerung nicht mehr verändert wurde. Der Vorteil in diesem Fall ist, dass die Seite nicht mehr auf den Sekundärspeicher zurückgeschrieben zu werden braucht. Dazu wird ein zusätzliches *dirty-Bit* zu jeder Seite in der Seitentabelle verwaltet, das bei jeder Schreiboperation gesetzt wird. Dadurch kann festgestellt werden, ob die Seite verändert wurde.

Übungsaufgabe 2.6 Welche Informationen benötigt das Betriebssystem z. B. in der Seitentabelle, um virtuellen Hauptspeicher zu realisieren?

`https://e.feu.de/1801-paging-info`

2.2 Leichtgewichtige Prozesse (Threads)

Zum Kontext eines Prozesses – das hatten wir in Abschnitt 2.1 gesehen – gehören neben den Registerinhalten auch Informationen über seinen Adressraum, bei seitenbasierter Speicherorganisation zum Beispiel eine Seitentabelle; vergleiche Abbildung 2.2. Die Verwaltung dieser Informationen kostet beim Prozesswechsel Rechenzeit.

Nun gibt es Probleme, bei deren Lösung man mehrere Prozesse einsetzen möchte, die quasi-parallel ablaufen und dabei alle auf denselben Speicherbereich zugreifen. Eine zeitaufwendige Einrichtung neuer Adressräume ist also beim Wechsel zwischen diesen Prozessen nicht erforderlich. Für solche Fälle

threads

hat man das Konzept der *leichtgewichtigen Prozesse* (threads = Fäden)[6] entwickelt, das in diesem Abschnitt vorgestellt wird.

Beispiel: Dateiserver

Betrachten wir zunächst ein Beispiel. In Rechnernetzen werden oft dedizierte *Dateiserver* (file server) verwendet, Rechner also, die darauf spezialisiert sind, auf Magnetplatten gespeicherte Dateien zu bearbeiten. Von jedem Rechner im Netz können Klienten Aufträge an den Server übermitteln, die dieser ausführt und beantwortet.

nur *ein* Prozess?

Weil der Dateiserver nur diese eine Aufgabe wahrnimmt, könnte man meinen, dass er im wesentlichen mit einem einzigen Anwendungsprozess auskommt. Dieser Prozess schaut in einem Briefkasten (mail box) nach, ob ein Auftrag A_i vorliegt. Wenn das so ist, wird ihm die logische Dateiadresse entnommen, und es wird zunächst geprüft, ob die entsprechenden Daten im Hauptspeicher-Cache des Servers stehen. Ist das nicht der Fall, wird die physische Adresse der Daten auf der Festplatte berechnet. Dann erteilt der Prozess einen Befehl an den Gerätetreiber.

Bei diesem Ansatz gibt es zwei Möglichkeiten: Der Serverprozess könnte jetzt blockieren, bis Gerätetreiber und Controller den Befehl ausgeführt haben;

[6] Das Wort *thread* = Faden sollte man nicht mit *threat* = Bedrohung verwechseln.

siehe Abschnitt 1.5. Dann könnte er die Antwort an den Klienten formulieren und abschicken. Dieses Verfahren ist recht einfach, aber sehr ineffizient; denn während der Serverprozess blockiert ist, wäre die CPU des Dateiservers die ganze Zeit über untätig. Oder aber der Serverprozess könnte eine Notiz über den Zustand der Bearbeitung von Auftrag A_i ablegen und schon einmal den nächsten Auftrag aus dem Briefkasten holen. Wenn dann später der Gerätetreiber den zu A_i gehörenden Befehl ausgeführt hat, könnte der Serverprozess mit der Bearbeitung des aktuellen Auftrags A_{i+j} innehalten und zunächst dem Klienten von Auftrag A_i seine Antwort schicken. Dieses verschachtelte Vorgehen versucht, mit einem einzelnen Prozess Parallelität nachzumachen; das ist zwar effizient, aber nicht so leicht zu programmieren und sehr unübersichtlich.

blockieren oder verschachteln?

Viel eleganter ist die Verwendung mehrerer Serverprozesse, von denen jeder einen kompletten Auftrag sequentiell ausführt. Während einer von ihnen blockiert, weil er auf die Erledigung eines Ein-/Ausgabebefehls wartet, braucht die CPU nicht untätig zu sein, denn inzwischen kann ja ein anderer Serverprozess rechnen.

mehrere Prozesse!

Diese Serverprozesse greifen alle auf dieselben Hauptspeicherbereiche zu: auf den Briefkasten und den Hauptspeicher-Cache. Es besteht also kein Grund, ihnen individuelle Adressräume zuzuweisen, die den Prozesswechsel verlangsamen. Deshalb verwendet man zur Steuerung eines Dateiservers am besten *leichtgewichtige Prozesse (Threads)*.

leichtgewichtige Prozesse, Threads

Mehrere Threads teilen sich ein Programm, einen Adressraum und dieselben Dateien. Jeder Thread hat aber seine eigenen Registerinhalte — insbesondere seinen eigenen Befehlszähler — und einen eigenen Stapel (Stacksegment). Solch eine Gruppe von zusammengehörigen leichtgewichtigen Prozessen wird als *Task* (Aufgabe) bezeichnet; siehe Abbildung 2.5.[7]

Task

Leichtgewichtige Prozesse können dieselben Zustände annehmen wie gewöhnliche Prozesse; sie können Kinder generieren und blockierende Systemaufrufe ausführen. Beim Zugriff auf den gemeinsamen Speicherbereich kann es zu den Problemen kommen, die wir in Abschnitt 2.3 beschreiben werden, so dass Synchronisationsmechanismen benötigt werden. Dabei kann man allerdings davon ausgehen, dass sich zusammengehörende leichtgewichtige Prozesse kooperativ verhalten.

Übungsaufgabe 2.7 Schlagen Sie eine Anwendung vor, die von der Benutzung von Threads profitiert, und eine, bei der dies nicht der Fall ist.

https://e.feu.de/1801-anwendungen-threads

[7] Diese Bezeichnungen werden aber nicht einheitlich verwendet; so heißen zum Beispiel in SunOS die Tasks selbst wieder *Prozesse* und es wird zwischen threads und LWPs unterschieden.

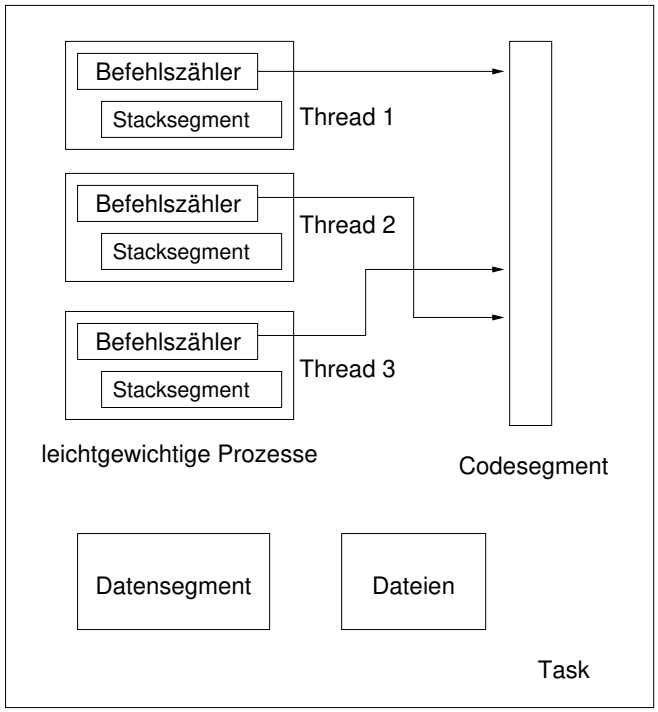

Abbildung 2.5: Drei Threads eines Tasks bei der quasi-parallelen Ausführung desselben Programmstücks; Codesegment, Datensegment und Dateien werden gemeinsam genutzt.

Implementierung
Kernel-Thread
Benutzer-Thread

Es gibt verschiedene Möglichkeiten, leichtgewichtige Prozesse zu implementieren: *Kernel-Threads* und *Benutzer-Threads*. Kernel-Threads werden *im Betriebssystemkern* realisiert, Benutzer-Threads hingegen im privaten Speicherbereich eines Prozesses. Dazu kommen noch Mischformen aus beiden Implementierungen.

im Betriebssystemkern

Bei einer Implementierung als Kernel-Threads im Betriebssystemkern werden die leichtgewichtigen Prozesse genau wie die schwergewichtigen behandelt; insbesondere wird jeder Prozesswechsel und das Scheduling vom Kern ausgeführt, wie in Abschnitt 1.4.1 beschrieben wurde. Diese Situation liegt zum Beispiel in Linux vor. Bei der Generierung eines leichtgewichtigen Prozesses verwendet man nicht den Befehl *fork*, wie wir ihn in Abschnitt 1.7 kennengelernt haben, sondern *clone*. Hierdurch wird ein Kindprozess erzeugt, der nicht nur sein Programm sondern auch seinen Speicherbereich vom Erzeugerprozess erbt.

clone

mit Bibliotheksprozeduren

Benutzer-Threads hingegen werden mit Hilfe von *Bibliotheksprozeduren* auf Benutzerebene implementiert. Wann immer ein Thread einen Systemaufruf ausführen möchte — zum Beispiel eine Semaphoroperation —, so ruft er stattdessen eine Bibliotheksprozedur auf. Sie entscheidet, ob der Thread suspendiert werden muss. Wenn das der Fall ist, vertauscht die Bibliotheksprozedur die aktuellen Registerinhalte mit denen eines anderen bereiten Threads, ohne dass der Betriebssystemkern involviert wird; das Betriebssystem weiß also gar nichts von der Existenz der Threads und behandelt den gesamten Task wie einen einzigen schwergewichtigen Prozess. So ergeben sich sehr kurze Umschaltzeiten

beim Wechsel zwischen den leichtgewichtigen Prozessen. Außerdem lässt sich das Scheduling der Threads bei diesem Ansatz vom Anwendungsprogrammierer steuern.

Diesen Vorteilen steht aber ein gravierender Nachteil gegenüber: Wenn ein leichtgewichtiger Prozess eines Tasks einen blockierenden Systemaufruf durchführt, wird der gesamte Task blockiert. Bei einer Implementierung leichtgewichtiger Prozesse im Betriebssystemkern könnte dagegen jetzt ein anderer Thread desselben Tasks rechnend gemacht werden.

Übungsaufgabe 2.8 Welche Art der Implementierung leichtgewichtiger Prozesse ist beim Dateiserver vorzuziehen?

`https://e.feu.de/1801-threads-implementierung`

Auch die nächste Aufgabe beschäftigt sich mit den Unterschieden bei der Implementierung leichtgewichtiger Prozesse.

Übungsaufgabe 2.9 Angenommen, Task T_1 enthält nur einen leichtgewichtige Prozess, und Task T_{100} enthält 100 Threads. Wieviel CPU-Zeit entfällt bei den beiden Implementierungsarten auf diese Tasks, wenn wir Round-Robin-Scheduling voraussetzen; vergleiche Abschnitt 1.4.1?

`https://e.feu.de/1801-cpu-threads`

2.3 Prozesssynchronisation

Aus Abschnitt 2.2 wissen wir, dass die Threads eines Prozesses das gemeinsame Datensegment benutzen können, um miteinander zu kommunizieren. Beispielsweise können sich Threads dort Nachrichten schreiben und lesen, und so miteinander kommunizieren.

Angenommen, in einem Rechnersystem finden in unregelmäßigen Abständen Ereignisse statt, über deren Häufigkeit Buch geführt werden soll. Ein Thread namens *Beobachter* zählt immer dann eine Variable *Zähler* hoch, wenn ein Ereignis stattgefunden hat. Ein zweiter Thread namens *Berichterstatter* druckt hin und wieder den Zählerstand aus und setzt die Variable auf Null zurück. Beide Threads existieren für immer.

```
Thread Beobachter;
begin
    repeat
        beobachte Ereignis;
        Zähler := Zähler + 1
    until false
end;
```

```
Thread Berichterstatter;
begin
    repeat
        print(Zähler);
        Zähler := 0
    until false
end;
```

gemeinsame Variable

Die Variable *Zähler* liegt in dem gemeinsamen Datensegment der Threads, auf den sie lesend und schreibend zugreifen können; vergleiche Abschnitt 2.1. Sie wird anfangs auf Null gesetzt.

Man sollte meinen, dass zu jedem Zeitpunkt die Summe aller ausgedruckten Zählerstände zusammen mit dem aktuellen Wert von *Zähler* die Anzahl aller Ereignisse angibt, die bisher stattgefunden haben.

Das stimmt aber nicht! Denn die beiden Threads laufen (quasi-)parallel ab, und dadurch kann es zum Beispiel zu folgender Ausführungsreihenfolge der Anweisungen kommen:

Wenn der Berichterstatter gerade den aktuellen Zählerstand ausgedruckt hat und in diesem Moment wegen Ablauf seiner Zeitscheibe unterbrochen wird, könnte der Beobachter rechnend werden. Angenommen, jetzt treten viele Ereignisse ein,[8] und die Variable *Zähler* wird entsprechend hochgezählt. Sobald der Berichterstatter wieder rechnend wird, setzt er als erstes den Zähler auf Null, und alle soeben aufgetretenen Ereignisse sind verloren.

Dieser Fehler hängt nicht davon ab, dass wir nur über eine CPU verfügen; er kann ebenso auftreten, wenn die Threads auf verschiedenen Prozessoren mit unbekannten Geschwindigkeiten ablaufen. Hier liegen sogenannte *Wettkampfbedingungen* (race conditions) vor; welcher Thread als erster ein bestimmtes Ziel erreicht, ist nicht vorhersehbar.

Wettkampfbedingungen

Die Fehlerursache liegt vielmehr darin, dass die beiden Anweisungen

```
print(Zähler);
Zähler := 0
```

kritischer Abschnitt

im Programm des Berichterstatters einen *kritischen Abschnitt* (critical section) bilden, der keine Unterbrechung durch den Beobachter verträgt. Ein kritischer Abschnitt ist ein Abschnitt im Programm, in dem

- gemeinsame Ressource wie z. B. Variable und Datenstrukturen benutzt werden,
- auf die mehrere Threads oder Prozesse lesend und schreibend zugreifen, so dass
- eine race condition entstehen kann.

Übungsaufgabe 2.10 Gibt es auch im Programm des Beobachters einen kritischen Abschnitt?

`https://e.feu.de/1801-kritischer-abschnitt`

Synchronisation

exklusiver Zugriff

Offenbar muss man die beiden Threads *synchronisieren,* um zu verhindern, dass beide zur gleichen Zeit in ihren kritischen Abschnitt eintreten. Also soll die Synchronisation den *exklusiven Zugriff* auf den kritischen Abschnitt garantieren.

[8] Wir stellen uns vor, dass alle Ereignisse gespeichert werden, die eintreten, während der Beobachter gerade auf die CPU wartet. Sie werden erfasst, sobald der Beobachter weiterrechnet.

Synchronisationsvariable

Eine naheliegende Möglichkeit besteht darin, eine *Synchronisationsvariable* namens *switch* zu verwenden, die wie ein Schalter den Zugang zu den kritischen Abschnitten regelt. Hat sie den Wert 0, so ist der Beobachter an der Reihe, beim Wert 1 darf der Berichterstatter rechnen:

```
Prozess Beobachter;
begin
    repeat
        while switch = 1 do no-op;
(* Beginn kritischer Abschnitt *)
        beobachte Ereignis;
        Zähler := Zähler + 1;
(* Ende kritischer Abschnitt *)
        switch := 1
    until false
end;
```

```
Prozess Berichterstatter;
begin
    repeat
        while switch = 0 do no-op;
(* Beginn kritischer Abschnitt *)
        print(Zähler);
        Zähler := 0;
(* Ende kritischer Abschnitt *)
        switch := 0
    until false
end;
```

Dabei steht *no-op* für *no operation*; der Thread Beobachter bleibt also in der *while*-Schleife und tut nichts, bis die Bedingung falsch wird, das heißt, bis *switch* den Wert 0 erhält. Dann führt er seinen kritischen Abschnitt aus und setzt *switch* auf 1, damit der Berichterstatter in seinen kritischen Abschnitt eintreten kann.

Man sieht schnell, dass diese Lösung zwei Nachteile aufweist:

busy waiting

- Beide Threads verbrauchen wertvolle CPU-Zeit, während sie in ihren *while*-Schleifen warten. Ein solches *geschäftiges Warten* (busy waiting) ist unerwünscht.

- Die beiden Threads können nur abwechselnd in ihre kritischen Abschnitte eintreten. Wenn einer von ihnen innerhalb des kritischen Abschnitts beendet werden sollte, kann der andere nie wieder den kritischen Abschnitt betreten.

Semaphor

Diese Schwierigkeiten lassen sich vermeiden, wenn man das Konzept des *Semaphors*[9] verwendet. Es wurde von Dijkstra[10] zur Lösung von Problemen entwickelt, bei denen mehrere Prozesse oder Threads ein Betriebsmittel belegen wollen, von dem insgesamt n Stück zur Verfügung stehen. Dabei kann es sich um n freie Speicherplätze handeln, um n CPUs oder um das Recht, in den kritischen Abschnitt eintreten zu dürfen. Im letzten Fall ist $n = 1$, weil ja zu einem Zeitpunkt immer nur *ein* Prozess seinen kritischen Abschnitt betreten darf.

[9] *Semaphor* kommt aus dem Griechischen und bedeutet Zeichenträger; die Bezeichnung wurde früher für Signalmasten benutzt, die zur optischen Übermittlung von Nachrichten dienten. Vergleiche auch *semaforo*, das italienische Wort für Verkehrsampel.

[10] Man spricht diesen Namen wie "deikstra".

Ein Semaphor S kann als abstrakter Datentyp[11] spezifiziert werden. Der Zustand von S besteht aus der Anzahl freier Betriebsmittel, gespeichert in einer Zählvariablen *count*, und einer Prozessmenge W. Falls $count \neq 0$, so ist W leer, ansonsten enthält W alle Prozesse, die sich bisher vergeblich um ein Betriebsmittel bemüht haben und darauf warten, dass wieder eines frei wird.

down

Auf S sind zwei Operationen definiert, *down* und *up*. Wenn ein Prozess ein Betriebsmittel benutzen will, ruft er die Operation *down* auf.

```
procedure down(S);
begin
    if S.count > 0
        then S.count := S.count − 1;
            (* aufrufender Prozess kann Betriebsmittel benutzen *)
        else (* alle Betriebsmittel belegt *)
            begin
                füge aufrufenden Prozess in die Menge S.W ein;
                blockiere aufrufenden Prozess
            end
end;
```

up

Wenn ein Prozess sein Betriebsmittel wieder freigibt, ruft er die Operation *up* auf. Sie bewirkt folgendes:

```
procedure up(S);
begin
    if S.W ist nicht leer
        then (* andere Prozesse warten; S.count ist 0 *)
            begin
                entferne einen Prozess aus S.W;
                mache ihn bereit
            end
        else
            S.count := S.count + 1
end;
```

Damit lässt sich das Problem vom Beobachter und Berichterstatter folgendermaßen lösen. Zunächst wird eine Semaphorvariable S definiert und ihre Zählvariable *count* auf 1 gesetzt. Die Variable *Zähler* erhält – wie oben – den Anfangswert 0.

[11] Siehe Abschnitt 1.3.3 zur den Begriffen Abstraktion und Kapselung und auch *Kurs Datenstrukturen I.*

```
Prozess Beobachter;
begin
    repeat
        down(S);
(* Beginn kritischer Abschnitt *)
        beobachte Ereignis;
        Zähler := Zähler + 1;
(* Ende kritischer Abschnitt *)
        up(S)
    until false
end;
```

```
Prozess Berichterstatter;
begin
    repeat
        down(S);
(* Beginn kritischer Abschnitt *)
        print(Zähler);
        Zähler := 0;
(* Ende kritischer Abschnitt *)
        up(S)
    until false
end;
```

Damit der Semaphor korrekt arbeitet, müssen die Operationen *down* und *up* in geeigneter Weise implementiert sein. So darf zum Beispiel bei einem Aufruf *down*(S) nach dem Test der Zählvariablen *count* kein anderer Prozess *down*(S) aufrufen, bevor der Wert von *count* — falls er positiv war — um eins heruntergezählt worden ist. Operationen, die nicht unterbrochen werden dürfen, nennt man *atomare* Operationen oder auch *unteilbare* Operationen; hiervon handelt die folgende Übungsaufgabe.

atomare Operation

unteilbare Operation

Übungsaufgabe 2.11 Begründen Sie, warum in der soeben beschriebenen Situation Fehler auftreten können.

https://e.feu.de/1801-semaphore

Wir stellen fest: Die Semaphor-Operationen *down* und *up* zur Realisierung kritischer Abschnitte enthalten selbst kritische Abschnitte! Haben wir also unser Problem nur verlagert? Ja; aber dadurch wird es leichter lösbar. Wenn man die Semaphor-Operationen nämlich im Betriebssystem implementiert, kann man während der Ausführung von *down* oder *up* alle Unterbrechungen sperren und damit den gerade beschriebenen Fehler verhindern; vergleiche Abschnitt 1.3.4.[12]

Unterbrechungen sperren

Falls die Semaphorvariable *count* nur die Werte 0 oder 1 annehmen kann, spricht man von einem *binären Semaphor*, der meistens dazu verwendet wird, den *wechselseitigen Ausschluss* (mutual exclusion) von zwei Prozessen sicherzustellen.

wechselseitiger Ausschluss

Beim folgenden *Erzeuger-Verbraucher-Problem* treten Semaphore auf, deren Zählvariablen größere Werte als 1 annehmen können. Angenommen, ein Erzeugerprozess E erzeugt bestimmte Objekte, die von einem Verbraucherprozess V verbraucht werden. Zum Beispiel kann E ein Compiler sein, der Anweisungen in Assemblersprache erzeugt, und V ein Assembler, der sie entgegennimmt und daraus Anweisungen im Binärcode macht. Der Erzeuger übergibt die Objekte nicht einzeln an den Verbraucher, sondern legt sie in einem

Erzeuger-Verbraucher-Problem

[12] Einem gewöhnlichen Anwenderprozess kann man dieses Recht natürlich nicht einräumen; das Sperren von Unterbrechungen ist also kein allgemein verfügbares Mittel zur Implementierung kritischer Abschnitte.

Puffer

Zwischenspeicher ab, einem sogenannten *Puffer* (buffer). Wann immer der Verbraucher ein Objekt verbraucht hat, holt er sich aus dem Puffer das nächste. Der Puffer kann maximal n Objekte speichern.
Hierbei treten folgende Synchronisationsprobleme auf:

- Erzeuger und Verbraucher sollten nicht gleichzeitig auf den Puffer zugreifen;
- der Erzeuger sollte nicht versuchen, ein Objekt in den vollen Puffer zu schreiben;
- der Verbraucher sollte nicht versuchen, ein Objekt aus dem leeren Puffer zu entnehmen.

Wir verwenden zur Lösung jedes Teilproblems einen eigenen Semaphor:

```
var Zugriff : semaphor;     (* binär                 *)
    Frei    : semaphor;     (* für n Betriebsmittel  *)
    Belegt  : semaphor;     (* für n Betriebsmittel  *)
begin                       (* Initialisierung       *)
    Zugriff.count := 1;
    Frei.count    := n;
    Belegt.count  := 0;
end;
```

```
Prozess Erzeuger;
begin
    erzeuge Objekt;
    down(Frei);
    down(Zugriff);
    lege Objekt in Puffer;
    up(Zugriff);
    up(Belegt)
end;
```

```
Prozess Verbraucher;
begin
    down(Belegt);
    down(Zugriff);
    entnimm ein Objekt aus Puffer;
    up(Zugriff);
    up(Frei);
    verbrauche Objekt
end;
```

Die Zählvariable *count* des Semaphors *Frei* wird in Pascal-Notation mit *Frei.count* bezeichnet; sie gibt an, wieviele Plätze im Puffer mindestens noch frei sind. Der Zählerstand von *Belegt* entspricht der Anzahl der mindestens belegten Plätze. Beachten Sie, dass die beiden Prozesse "über Kreuz" symmetrisch sind. Das Programm löst die obigen drei Synchronisationsprobleme. Bevor ein Erzeuger ein neues Objekt in den Puffer legt, muss er prüfen, dass der Puffer nicht schon voll ist, indem er eine *down*-Operation auf *Frei* ausführt. Wenn der Puffer voll ist, muss er sich in die Warteschlange von *Frei* stellen. Wenn nicht, dann kann der Erzeuger versuchen, auf den Puffer zuzugreifen. Er führt eine *down*-Operation auf *Zugriff* aus, damit er warten muss, wenn gerade ein paraleller Zugriff auf den Puffer stattfindet. Nachdem das erzeugte

Objekt in den Puffer gelegt wird, führt der Erzeuger eine *up*-Operation auf *Zugriff* aus, um den exklusiven Zugriff wieder freizugeben. Danach wird noch eine *up*-Operation auf *Belegt* ausgeführt, da jetzt ein Objekt mehr im Puffer liegt. Bevor der Verbraucher auf den Puffer zugreift, muss er zuerst testen, ob der Puffer leer ist. Er führt eine *down*-Operation auf *Belegt* aus. Wenn der Puffer leer ist, dann stellt er sich in die Warteschlange von *Belegt*. Wenn nicht, kann er auf den Puffer zugreifen. Nachdem der Verbraucher den exklusiven Zugriff freigegeben hat, führt er eine *up*-Operation auf *Frei* aus, da der Puffer einen freien Platz mehr hat.

Übungsaufgabe 2.12 Hätte man bei den Prozessen Erzeuger und Verbraucher ebensogut die Aufrufe *down*(Zugriff) ganz vorn und *up*(Zugriff) ganz am Schluss durchführen können?

`https://e.feu.de/1801-up-down`

Ein berühmtes Problem der Prozess-Synchronisation ist das auf Dijkstra zurückgehende Problem der dinierenden Philosophen. Es lautet wie folgt: $n \geq 2$ Philosophen sitzen an einem runden Tisch. Jeder dieser n Philosophen durchläuft zyklisch die drei Zustände „Denken“, „Hungrig“ und „Essen“. Um essen zu können, braucht jeder Philosoph gleichzeitig ein links und ein rechts von seinem Teller liegendes Essstäbchen. Den n Philosophen stehen aber nur insgesamt n Stäbchen zur Verfügung (zwei an dem runden Tisch aneinander angrenzende Teller sind durch genau ein Stäbchen getrennt, vgl. Abbildung 2.6). Wenn also zwei benachbarte Philosophen gleichzeitig hungrig werden und dann essen wollen, so wird es Schwierigkeiten geben[13].

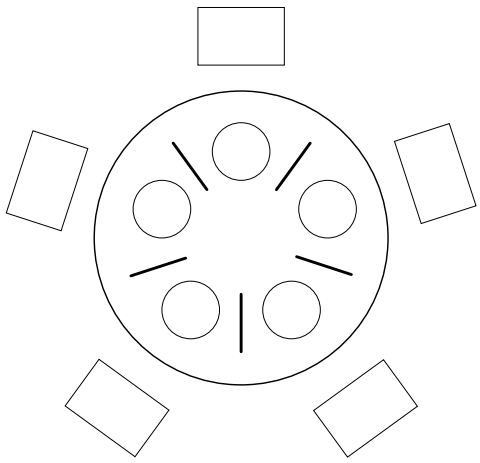

Abbildung 2.6: Dinierende Philosophen.

[13] von Problemen hygienischer Art einmal abgesehen

Betrachten wir jeden Philosophen als Prozess, so wird es gerade darauf ankommen, das gleichzeitige Essen zweier benachbarter Philosophen (oder auch nur den Versuch dazu) zu vermeiden.

Die einfachste Form einer Lösung dieses Problems wäre die Einführung von Semaphoren $S_0, \ldots, S_{n-1}$ für jedes Stäbchen. Für jeden Philosophen i liefere links(i) (rechts(i)) den Index des Semaphors für das Stäbchen links (rechts) vom ihm. Alle Semaphore werden mit 1 initialisiert.

Für den i-ten Philosophen könnte eine Lösung wie folgt lauten:

repeat
  denken;
  hungrig;
  down($S_{\text{links(i)}}$); (* versuche, das linke Stäbchen zu nehmen *)
  down($S_{\text{rechts(i)}}$); (* versuche, das rechte Stäbchen zu nehmen *)
  essen;
  up($S_{\text{rechts(i)}}$); (* lege das rechte Stäbchen zurück *)
  up($S_{\text{links(i)}}$); (* lege das linke Stäbchen zurück *)
until false;

Übungsaufgabe 2.13 Welche Blockade kann bei dieser einfachen Lösung entstehen?

https://e.feu.de/1801-blockade

Man löst das Problem dieser Blockade, indem ein Philosoph nur dann Stäbchen aufnehmen darf, wenn beide frei sind. Dazu braucht man einen Semaphor *mutex*,[14] der mit 1 initialisiert wird. Bevor ein Philosoph versucht, die Stäbchen zu nehmen oder zurückzulegen, führt er eine *down*-Operation auf *mutex* aus. Nachdem er die Stäbchen genommen oder zurückgelegt hat, führt er eine *up*-Operation auf *mutex* aus. Es kann also immer nur ein Philosoph gleichzeitig Stäbchen nehmen oder zurücklegen. Kann aber ein Philosoph nicht beide Stäbchen aufnehmen, so wird er schlafen gelegt. Das wird realisiert, indem jeder Philosoph i einen Semaphor $p[i]$ hat, der mit 0 initialisiert wird. Das Ziel des Semaphors $p[i]$ ist, den i-ten hungrigen Philosoph zu blockieren, d. h. warten zu lassen, falls eins seiner beiden Stäbchen nicht frei ist. Jeder Philosoph i benötigt noch eine Statusvariable $status[i]$, um zu verfolgen, ob er gerade hungrig ist oder denkt oder isst. Ein Philosoph kann nur in den Zustand von Essen übergehen, wenn seine beiden Nachbarn nicht beim Essen sind. Dieser $status[i]$ wird zu Anfang auf Denken gesetzt.

[14] Mutex steht für mutual exclusion.

Folgender Prozess beschreibt den Philosophen i:

```
repeat
  denken;
  stäbchen_nehmen(i);
  essen;
  stäbchen_weglegen(i);
until false;
```

Die Prozedur stäbchen_nehmen(i):

```
down(mutex);
status[i] := Hungrig;
teste(i); (* nimmt beide Stäbchen, wenn sie frei sind *)
up(mutex);
down(p[i]); (* hier schläft der Philosoph ein, wenn er nicht beide
               Stäbchen bekommen hat *)
```

Die Prozedur stäbchen_weglegen(i):

```
down(mutex);
status[i] := Denken;
teste(links(i)); (* evtl. Nachbarn aufwecken *)
teste(rechts(i)); (* evtl. Nachbarn aufwecken *)
up(mutex);
```

Zum Schluss die Prozedur teste(i):

```
if (status[i] = Hungrig
    and status[links(i)] <> Essen
    and status[rechts(i)] <> Essen)
then begin
  status[i] := Essen;
  up(p[i]);
end
```

Die Prozedur teste(i) stellt fest, ob der Philosoph i hungrig ist und die zwei Stäbchen bekommen kann. Wenn ja, dann geht er in den Zustand „Essen“ über. Man beachten, dass die Prozedur teste sowohl von Philosophen für sich selbst aufgerufen wird (in stäbchen_nehmen) als auch für seine Nachbarn (in stäbchen_weglegen). Außerdem wird es hier nicht automatisch „gerecht“ zugehen, was die Verteilung der Essenszeiten angeht, wie man sich leicht überlegen kann.

2.4 Dateisysteme

Neben den Prozessen sind *Dateien* und *Dateisysteme* die Objekte, mit denen jeder Anwender und Programmierer zu tun hat; sie bestimmen zu einem großen Teil seine Arbeitsumgebung.

Wir wollen uns im folgenden Abschnitt zunächst mit der *logischen Sicht* auf die Dateien beschäftigen, die ein Dateisystem den Benutzern bietet. Dabei interessieren uns besonders *Verzeichnisse* und *Pfade*, wie UNIX sie verwendet, und die Befehle zu ihrer Verwaltung.

In Abschnitt 2.4.2 diskutieren wir dann, wie das Dateisystem intern die logische Sicht der Dateien auf die physische Realität des Sekundärspeichers abbildet.

2.4.1 Dateien, Verzeichnisse und Pfade

Datei

Dateisysteme dienen der Verwaltung von *Dateien* (files). Eine Datei ist – wie schon in Abschnitt 1.6 erwähnt – eine Folge von Datensätzen, die zusammengehörige Information enthalten. Für den Benutzer wird eine Datei durch ihren

Name

Namen (filename) kenntlich. Es empfiehlt sich, *sprechende Namen* zu vergeben, aus denen der Inhalt der Dateien ersichtlich wird.[15]

Typ

Oft wird der *Typ* einer Datei durch eine *Erweiterung* (extension), auch *Suffix* genannt, des Dateinamens bezeichnet. Wer zum Beispiel mit dem Formatierer LaTeX arbeitet, kennt wohl Dateien der Art

SeminarArbeit.tex SeminarArbeit.pdf SeminarArbeit.aux,

die – in dieser Reihenfolge – einen zu formatierenden Text namens SeminarArbeit, den geräteunabhängig als PDF formatierten Text zum Anzeigen und Drucken und die automatisch generierten Hilfsinformationen zum Formatieren enthalten.

Verzeichnis

In UNIX sind Dateien in *Verzeichnissen* (directories) zusammengefasst. Der Benutzer befindet sich zu jedem Zeitpunkt in einem aktuellen Arbeitsverzeichnis, dessen Namen man sich mit dem Befehl *pwd* (*p*rint *w*orking *d*irectory) ausgeben lassen kann.[16]

pwd

Ein Verzeichnis kann Dateien und weitere Unterverzeichnisse enthalten. Welche Objekte im aktuellen Verzeichnis enthalten sind, kann man sich mit dem Befehl *ls* auf*li*sten lassen; durch Angabe weiterer Parameter lässt sich festlegen, welche *Attribute* der im Verzeichnis enthaltenen Objekte ausgegeben

Attribute

[15] Man kann dabei den Anregungen für die Benennung von Variablen aus dem Kurs *Konzepte imperativer Programmierung* folgen.

[16] Alle im folgenden aufgeführten Befehle sind, soweit nicht anders erwähnt, Systemaufrufe in UNIX, die in dieser Form auch an der Benutzerschnittstelle zur Verfügung stehen. Um eine genaue Erklärung eines Befehls *command* und seiner Varianten nachzulesen, kann man den Befehl *man command* aufrufen.

werden. So kann zum Beispiel der Befehl *ls -ls* eine Ausgabe folgender Art erzeugen: *ls*

```
 1  drwxrwxr--  2 mueller  bteam   1024 Jul 27 16:43  archiv
 4  -rw-------  2 mueller  bteam   4038 Jul 27 18:46  entwurf.tex
30  -rw-r-----  1 fischer  bteam  29710 Mär 20  2015  flip
 2  -rwxr-xr--  1 mueller  bteam   1730 Jul 15 10:53  myprogram
 1  drwx------  4 mueller  bteam   1024 Mai 12 09:01  privat
```

Ganz rechts stehen die *Namen* der fünf in diesem Verzeichnis enthaltenen Objekte; nach ihnen ist die Ausgabe alphabetisch sortiert.[17] Name

In der Spalte davor ist aufgelistet, wann zum letzten Mal schreibend auf diese Objekte *zugegriffen* worden ist. Liegt dieser Zeitpunkt in einem vergangenen Jahr, so wird statt der Uhrzeit das Jahr angegeben. Das Datum des letzten *lesenden Zugriffs* liefert *ls -lsu*. letzter Zugriff

Links vom Datum steht die *Größe* des jeweiligen Objekts in Bytes. Die Größe in Blöcken, wie sie in Abschnitt 1.2.3 eingeführt wurden, wird in der Spalte ganz links angegeben, dafür sorgt der Parameter *s* des *ls*-Kommandos, bei *ls -l* wird diese Spalte nicht mit ausgegeben. Größe Blöcke

Übungsaufgabe 2.14 Können Sie bei diesem Beispiel die Blockgröße in Byte bestimmen? Wie erklären sich die Abweichungen?

`https://e.feu.de/1801-blockgroesse`

Die nächsten Eintragungen haben mit den *Zugriffsberechtigungen* auf die Objekte zu tun. Hierzu ein paar Vorbemerkungen. Auf einem Rechner mit mehreren Benutzern kann man nicht jedem Benutzer den Zugriff auf alle überhaupt vorhandenen Daten erlauben; in einem solchen System ließe sich kein wirksamer *Datenschutz* realisieren, und die Gefahr der versehentlichen Zerstörung von Information wäre zu groß. Es ist aber auch nicht sinnvoll, wenn jeder Benutzer nur auf seine eigenen Dateien zugreifen darf, denn das beschränkt die Möglichkeit zur Kooperation. Benötigt werden also Mechanismen zur Vergabe abgestufter Zugriffsrechte. Naheliegende Ansätze wären, zu jeder Datei eine Liste aller Benutzer anzulegen, die Zugriff auf die Datei haben, oder für jeden Benutzer eine Liste mit allen Dateien, auf die er zugreifen darf. Beide Verfahren sind recht ineffizient. Zugriffsrechte

In UNIX wird deshalb ein anderer Weg beschritten, um die Zugriffsrechte auf eine Datei oder ein Verzeichnis festzulegen. Man teilt die Systembenutzer in drei Klassen ein: den Eigentümer des Objekts – er wird in diesem Zusammenhang als *u*ser bezeichnet –, die Arbeitsgruppe (*g*roup), der er angehört, und alle übrigen Systembenutzer (*o*ther). In der Ausgabe des Befehls *ls -ls* steht in der vierten Spalte von links der Name des Besitzers, in der fünften der Name seiner Arbeitsgruppe. Benutzerklassen

[17] Strenggenommen haben wir nur die *sichtbaren* Objekte auflisten lassen. In jedem Verzeichnis gibt es auch unsichtbare Objekte; will man sie sehen, muss man *ls -lsa* eingeben.

Zugriffsart

Außerdem unterscheidet man zwischen den *Zugriffsarten* Lesen (*r*ead), Schreiben (*w*rite) und Ausführen (e*x*ecute). Das Recht auf Schreibzugriff (write permission) erlaubt dabei auch ein Überschreiben und enthält deshalb das Recht zum Löschen des Objekts. Bei einem Verzeichnis bedeutet das Ausführungsrecht, dass man dieses Verzeichnis zum aktuellen Arbeitsverzeichnis machen darf.

Nun kommen wir zur Ausgabe der Objekte in unserem Beispielverzeichnis zurück und betrachten die zweite Spalte von links. Das erste Zeichen bei den Objekten *archiv* und *privat* ist ein *d* für *d*irectory — hier handelt es sich also um Verzeichnisse. Bei den übrigen Objekten finden wir an dieser Stelle einen Strich; dies sind Dateien.

Die in der zweiten Spalte auf das erste Zeichen folgenden 9 Zeichen beschreiben die Zugriffsberechtigungen für die verschiedenen Benutzerklassen in der Reihenfolge

```
rwxrwxrwx
 u  g  o
```

Wo ein Strich steht, ist das betreffende Recht nicht vergeben. So hat zum Beispiel die vierte Zeile

```
-rwxr-xr--   1 mueller  bteam   1730 Jul 15 10:53  myprogram
```

folgende Bedeutung: Der Eigentümer Müller hat an seiner Datei *myprogram* das Lese-, Schreib,- und Ausführungsrecht. Die übrigen Mitglieder seiner Arbeitsgruppe *bteam* dürfen das Programm lesen und ausführen, aber nicht schreiben. Alle übrigen Benutzer dürfen das Programm zwar lesen, aber weder ausführen noch schreiben. Und die Zeile

```
drwxrwxr--   4 mueller  bteam    512 Jul 27 16:43  archiv
```

besagt, dass der Eigentümer und alle übrigen Gruppenmitglieder im Verzeichnis *archiv* Einträge von Objekten lesen und schreiben dürfen und dieses Verzeichnis auch zum aktuellen Arbeitsverzeichnis machen dürfen; hierzu verwendet man den Befehl *cd archiv*, wobei *cd* für *c*hange *d*irectory steht. Alle übrigen Benutzer dürfen aber nur lesen, welche Objekte im Verzeichnis *archiv* enthalten sind.

cd

Übungsaufgabe 2.15 Kann Benutzer Müller die Datei *flip* editieren, wenn Müller und Fischer Mitglieder der Gruppe *bteam* sind?

`https://e.feu.de/1801-zugriffsrechte-gruppe`

Angenommen, Müller möchte allen Mitgliedern seiner Arbeitsgruppe gestatten, an seinem Programm *myprogram* mitzuschreiben. Dann kann er den Befehl

chmod

chmod g+w myprogram

erteilen. Hierbei steht *chmod* für *ch*ange *mod*e, und *g+w* bedeutet, dass der Gruppe das Schreibrecht hinzugefügt wird; mit *g-x* hätte Müller ihr das Recht zum Ausführen des Programms entzogen. Außer dem *Superuser* kann nur der Eigentümer eines Objekts dessen Rechte ändern.

Übungsaufgabe 2.16 Was bewirkt der Befehl *chmod go+rw entwurf.tex*?

`https://e.feu.de/1801-chmod`

Rechte an einem Objekt vererben sich übrigens nicht automatisch auf kleinere Benutzerklassen: Im Beispiel

```
----r--r--   1 meier  cteam    402 Jul 27 16:43  akte
```

hat jeder Benutzer das Leserecht an der Datei *akte* – nur ihr Eigentümer Meier nicht! Meier kann sich jedoch das Leserecht wieder zuweisen.

keine Vererbung

Wer im aktuellen Verzeichnis schreibberechtigt ist, kann dort mit *touch neudatei* oder *mkdir neuverzeichnis* eine neue Datei mit 0 Byte Länge beziehungsweise ein neues Verzeichnis anlegen; hierbei steht *mk* für *mak*e.[18] Der Befehl *rm neudatei* löscht die Datei *neudatei* wieder; natürlich bedeutet *rm* hier *rem*ove. Zum Löschen eines leeren Verzeichnisses verwendet man *rmdir*. Mit dem Befehl *rm -r neuverzeichnis* kann man das Verzeichnis *neuverzeichnis* und alle darin enthaltenen Objekte löschen, selbst wenn man an diesen nicht schreibberechtigt ist! (Vorsicht mit dieser Option!)

Objekte anlegen und löschen

touch, mkdir

rm, rmdir

Mit dem Befehl *cp altdatei neudatei* kann man eine Datei, die man lesen darf, kopieren (*cop*y) und dabei den Namen der Kopie festlegen. Dabei wird man zum Eigentümer der Kopie. Man kann mit Varianten dieses Befehls auch Verzeichnisse kopieren.

Datei kopieren

cp

Übungsaufgabe 2.17 Kann in unserem Beispiel Benutzer Müller sich eine editierbare Version der Datei *flip* verschaffen?

`https://e.feu.de/1801-edit-recht`

[18] Meistens werden aber neue Dateien nicht von Hand, sondern von System- oder Anwendungsprogrammen erzeugt.

Wenn man in Verzeichnissen Unterverzeichnisse anlegt, entstehen *baumförmige* Strukturen; ein Beispiel ist in Abbildung 2.7 zu sehen.

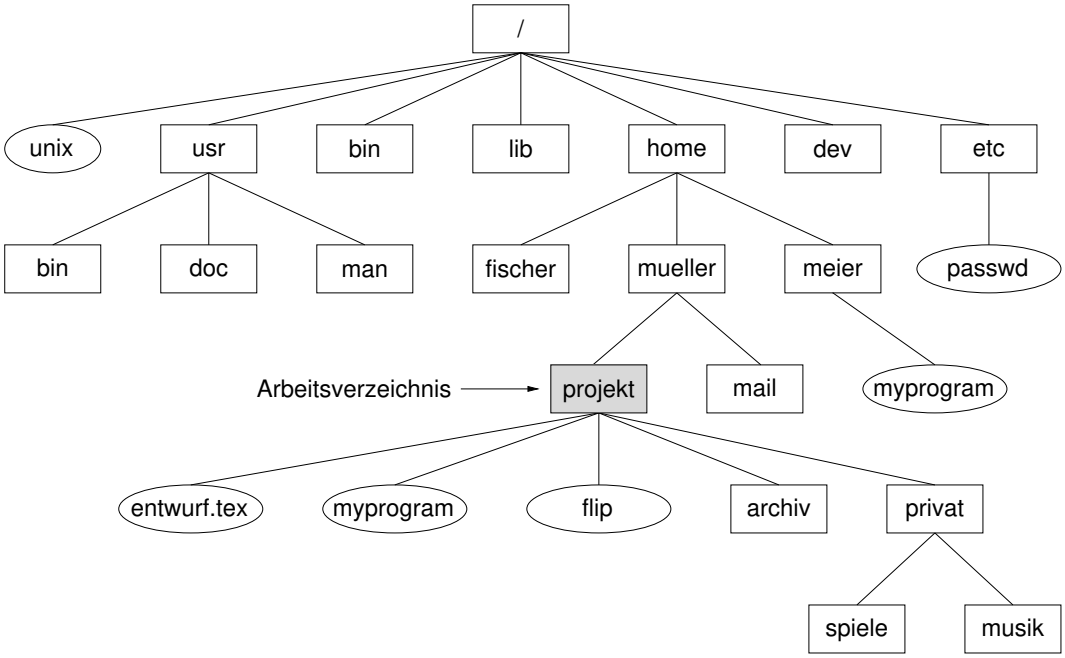

Abbildung 2.7: Ein Ausschnitt aus einer Verzeichnisstruktur unter UNIX.

Verzeichnisbaum

Ganz oben befindet sich das Wurzelverzeichnis (root), das immer mit dem Schrägstrich (slash) / bezeichnet wird. Darin sind unter anderem eine Datei *unix* mit dem Betriebssystemkern[19] enthalten und ein Verzeichnis *home* mit Benutzerdateien. Weiter unten sehen wir im Verzeichnis *mueller* ein Verzeichnis namens *projekt*, dessen Inhalt wir bei unserem Beispiel oben aufgelistet hatten.

home directory

Jedem Benutzer wird vom Systemverwalter ein *Heimatverzeichnis* (home directory) zugewiesen, das bei jeder Anmeldung (log-in) zunächst das aktuelle Arbeitsverzeichnis ist. Für den Benutzer Müller könnte das zum Beispiel das Verzeichnis *mueller* sein. Mit dem Befehl *cd projekt* kann er sich nach der Anmeldung in das Verzeichnis *projekt* begeben.

Pfad

Wir sehen, dass sowohl Meier als auch Müller eine Datei namens *myprogram* besitzen. Dies führt aber nicht zu Verwechselungen, denn für das Filesystem sind die *Zugriffspfade* im Verzeichnisbaum entscheidend — und die sind bei diesen beiden Dateien verschieden! So hat Meiers Datei den vollständigen Namen

/home/meier/myprogram,

während die vollständige Bezeichnung für Müllers Datei

/home/mueller/projekt/myprogram

absolute und relative Pfadnamen

lautet. Neben diesen *absoluten Pfadnamen* können die Benutzer auch *relati-*

[19] Unter Linux ist der Betriebssystemkern oft unter */vmlinuz* oder */boot/vmlinuz* zu finden.

ve Pfadnamen verwenden, die im aktuellen Arbeitsverzeichnis beginnen. So kann Müller seine Datei einfach durch *myprogram* ansprechen, während er die gleichnamige Datei von Benutzer Meier

../../meier/myprogram

nennen kann; dabei bezeichnet ".." stets das Elternverzeichnis. Für das aktuelle Verzeichnis gibt es übrigens die Bezeichnung ".", und das Heimatverzeichnis wird mit "∼" bezeichnet.

".." , "." und "∼"

Diese Pfadnamen werden auch bei zahlreichen Befehlen verwendet. Will zum Beispiel Müller seine Datei *entwurf.tex* an Meier abgeben, so kann er den Befehl *mv* für *move* dazu verwenden und

mv

mv entwurf.tex ../../meier

eingeben.[20] Voraussetzung ist, dass Müller im Verzeichnis *meier* schreibberechtigt ist. Name, Eigentümer und Zugriffsrechte der Datei bleiben hierbei erhalten. Damit Meier mit der Datei *entwurf.tex* etwas anfangen kann, sollte Müller vorher zumindest allen Benutzern seiner Gruppe Schreib- und Leserecht gewähren; siehe Übungsaufgabe 2.16.

Um zu erreichen, dass Meier und Müller beide bequem an der Datei *entwurf.tex* arbeiten können, gibt es noch eine andere Möglichkeit: Müller setzt zunächst die Zugriffsrechte um, wie gerade besprochen. Dann gibt er den Befehl

ln entwurf.tex ../../meier

ein. Hierbei steht *ln* für *link* und bewirkt, dass im Verzeichnis *meier* ein weiterer Verweis (*hard link*) auf die Datei *entwurf.tex* angelegt wird. Dieser Verweis heißt hier dann ebenfalls *entwurf.tex*, könnte aber auch einen anderen Namen bekommen. Wenn einer von beiden an der Datei Änderungen vornimmt, sind sie auch für den anderen sichtbar.[21] Wenn Meier oder Müller seinen Eintrag *entwurf.tex* löscht, bleibt die Datei für den anderen erhalten, da noch ein anderer Verweis auf diese Datei besteht.

ln

hard link

Damit können wir in unserer Beispielausgabe des Befehls *ls -ls* auch die dritte Spalte von links erklären: Sie gibt die Anzahl der Verweise auf das betreffende Objekt an. Bei Dateien beträgt diese Zahl eins plus die Anzahl der zusätzlich eingerichteten Verweise; bei *entwurf.tex* sehen wir deshalb eine 2. Bei Verzeichnissen verweist zusätzlich jedes Verzeichnis auf sich selbst (mit ".") und auf sein Elternverzeichnis (mit ".."). Deshalb steht bei dem leeren Verzeichnis *archiv* eine 2 und bei *privat* die Zahl 4, weil *privat* noch zwei Unterverzeichnisse enthält, vergleiche Abbildung 2.7.

[20] Eine Kopie der Datei behält Müller dabei nicht; die müsste er sich vorher eigens anfertigen.

[21] Diese Arbeitsweise soll hier aber nicht empfohlen werden, da sie auch erhebliche Nachteile mitbringt. Für kooperatives Arbeiten gibt es viel bessere Unterstützungen, z. B. Versionskontrolle oder Synchronisation von Dateien.

2.4.2 Interne Struktur von Dateisystemen

In Abschnitt 2.4.1 haben wir uns damit beschäftigt, welche *logische* Sicht auf die Dateien das Dateisystem als der für die Ein- und Ausgabe zuständige Teil des Betriebssystems den Benutzern bietet. Jetzt wollen wir untersuchen, wie das Dateisystem intern die logische Sicht auf die *physische* Sicht abbildet.

Block

Während eine Datei sich aus logischer Sicht als eine Folge von Datensätzen darstellt, ist sie aus physischer Sicht eine Folge von gleich großen *Blöcken*. Auf der Magnetplatte wird jeder Block mit Zusatzinformation versehen und in einem Sektor gespeichert; vergleiche Abschnitt 1.2.3 und Übungsaufgabe 2.14.

zusammenhängende Speicherung

Aus zwei Gründen wäre es wünschenswert, die Blöcke einer Datei möglichst hintereinander[22] auf der Platte zu speichern: Bei sequentiellem Zugriff auf mehrere aufeinander folgende Blöcke wird dadurch die Zeit für die Bewegung der Schreib-/Leseköpfe minimiert. Und bei wahlfreiem Zugriff auf einzelne Blöcke kann man leicht die Blocknummern berechnen: Wenn der $i-$te Block der Datei gelesen werden soll und die Datei bei Block b beginnt, so muss der Gerätetreiber einen Leseauftrag für den Block mit der Nummer $b+i$ erhalten.

externe Fragmentierung

Leider führt der Wunsch nach zusammenhängender Speicherung von Dateien zu demselben Problem der *externen Fragmentierung*, das wir in Abschnitt 2.1 bei der Hauptspeichervergabe beobachtet haben: Zwischen den Dateien entstehen Lücken, die sich nicht mehr zur Speicherung längerer Dateien eignen. Eine Kompaktifizierung durch Zusammenschieben der vorhandenen Dateien ist zwar möglich, wird aber wegen des hohen Zeitaufwands in der Regel nicht bei laufendem Betrieb durchgeführt.

nichtzusammenhängende Speicherung

Man hat deshalb auch beim Sekundärspeicher die Forderung nach zusammenhängender Speicherung aufgegeben und stattdessen Speicherverfahren entwickelt, bei denen die Blöcke einzeln gespeichert werden können, wo immer gerade Platz frei ist. Hier stellt sich die Frage, wie man die Blöcke effizient wiederfindet.

verkettete Liste

Eine naive Lösung könnte darin bestehen, die Blöcke einer Datei in einer *Liste* zu *verketten:* In dem Verzeichnis, das die Datei enthält, wäre dann beim Dateinamen die physische Adresse von Block Nr. 0 der Datei aufgeführt; am Schluss dieses Blocks stünde die physische Blocknummer von Dateiblock Nr. 1, und so fort. Diese Idee geht zwar effizient mit dem Speicherplatz um und vermeidet externe Fragmentierung, sie hat aber einen anderen schwerwiegenden Nachteil: Wahlfreier Zugriff wird nicht unterstützt. Um Block Nr. i zu lesen, sind nämlich i Zugriffe auf den externen Speicher auf alle Blöcke 0 bis $i-1$ notwendig.

kein wahlfreier Zugriff

FAT

Die Betriebssysteme MS-DOS und OS/2 umgingen diesen Nachteil durch einen einfachen Trick: Verkettet werden nicht die Blöcke, sondern ihre physischen Blockadressen. Abbildung 2.8 zeigt ein Beispiel für diese als *file-allocation table* bekannte Struktur. Sie wird als

$$\text{FAT} : \text{array}[0..\text{MaxBlockNr}-1] \text{ of BlockNr}$$

[22] *Hintereinander* bedeutet bei einer Festplatte: auf benachbarten Sektoren oder — falls das nicht möglich ist — auf benachbarten Zylindern.

implementiert und enthält für jeden Block der Magnetplatte[23] einen Eintrag. Im Dateiverzeichnis steht – wie oben – bei jedem Dateinamen *myprogram* die physische Adresse i von Dateiblock Nr. 0. Der Eintrag FAT[i] enthält dann die Adresse j des zweiten Blocks der Datei *myprogram*, in FAT[j] steht die Adresse des dritten Blocks, und so fort. Für den letzten Block l der Datei hat FAT[l] den speziellen Wert *eof*, der für *e*nd *o*f *f*ile steht. Für freie Blöcke lautet der Eintrag *free*. Wenn eine Datei verlängert werden soll, kann man also die FAT dazu benutzen, einen unbelegten Block zu finden.

eof

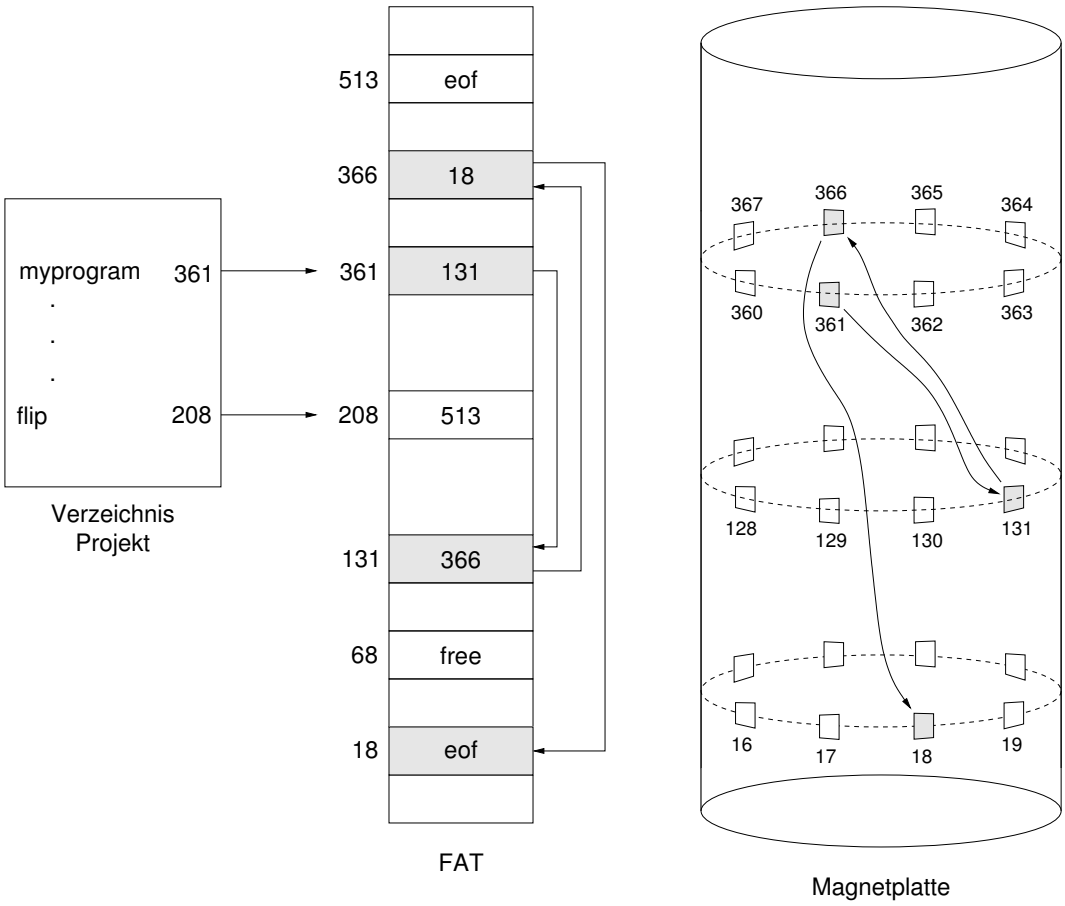

Abbildung 2.8: Ein Beispiel für eine *file-allocation table* und die zugehörige Verteilung der Dateien auf der Festplatte.

Die FAT wird auf konsekutiven Blöcken im Externspeicher abgelegt. Die Blockgröße sollte so bemessen sein, dass die FAT zur Laufzeit in den Cache im Hauptspeicher passt. Dann kann man die physische Adresse des i-ten Blocks einer Datei mit i Hauptspeicherzugriffen herausfinden, also rund tausendmal schneller als bei einer Verkettung der Blöcke im Externspeicher.

[23] Oft werden die Platten noch in Partitionen oder Filesysteme unterteilt; dann gibt es zu jedem dieser Teilsysteme eine eigene FAT.

Übungsaufgabe 2.18 Wieviel Platz belegt die file-allocation table einer Partition von 32 MByte bei einer Blockgröße von 512 Byte? Können Sie eine allgemeine Formel aufstellen, die den Platzbedarf der FAT in Blöcken angibt?

`https://e.feu.de/1801-fat-platzbedarf`

Index

Anstatt die Blöcke einer Datei als verkettete Liste zu verwalten, kann man für jede Datei einen *Index* anlegen. Ein Index ist eine Tabelle, die zu jeder logischen Blocknummer die zugehörige physische Blocknummer enthält. Sie entspricht der Seitentabelle eines Prozesses bei der Hauptspeicherverwaltung; vergleiche Abbildung 2.2. Diese Indextabelle wird selbst auch im Externspeicher abgelegt.

Auch mit diesem Ansatz ist das Problem der externen Fragmentierung beseitigt. Heikler ist dagegen die Frage nach der Indexgröße: Bei einer sehr kurzen Datei ist es nicht zu rechtfertigen, einen ganzen Block für die Speicherung des Index zu verwenden. Wenn dagegen die Datei sehr lang ist, reicht ein einzelner Block hierfür nicht aus; es kann dann sogar notwendig werden, einen Index für den Index anzulegen.

inode

In UNIX wird eine recht elegante Variante des Indexprinzips verwendet. Für jede Datei und für jedes Verzeichnis gibt es eine Struktur, die als *inode*[24] bezeichnet wird; ein Beispiel sehen Sie in Abbildung 2.9.

Am Anfang eines inode stehen die Attribute des Objekts, die wir uns in Abschnitt 2.4.1 mit dem Befehl *ls -ls* angesehen hatten, wie Zugriffsrechte, Eigentümer und Gruppe, Zeitstempel, Größe und Anzahl der Verweise auf das Objekt. Es folgen die physischen Adressen der ersten 12 Blöcke[25] der Datei. Dann kommt die Adresse eines Blocks, der die Adressen der nächsten logischen Dateiblöcke enthält – ein einfach-indirekter Index. Daran schließen die Adressen eines zweifach-indirekten und schließlich eines dreifach-indirekten Indexblocks an; dieser enthält die Adressen von zweifach-indirekten Indexblöcken, von denen jeder die Adressen einfach-indirekter Indexblöcke enthält.

mehrfach-indirekte Indices

Übungsaufgabe 2.19 Angenommen, wir haben eine Partition von 32 MByte bei einer Blockgröße von 512 Byte. Wie viele Blöcke darf eine Datei haben, damit sie sich durch einen inode beschreiben lässt?

`https://e.feu.de/1801-inode-blockzahl`

[24] Hierbei steht *inode* für *i-node* = Index-Knoten; die deutsche Form ist aber ungebräuchlich.

[25] Diese Zahl variiert je nach UNIX-Variante. 12 Blöcke gilt u. a. für das Ext2-System von Linux. In System V waren z. B. 10 Blöcke gebräuchlich.

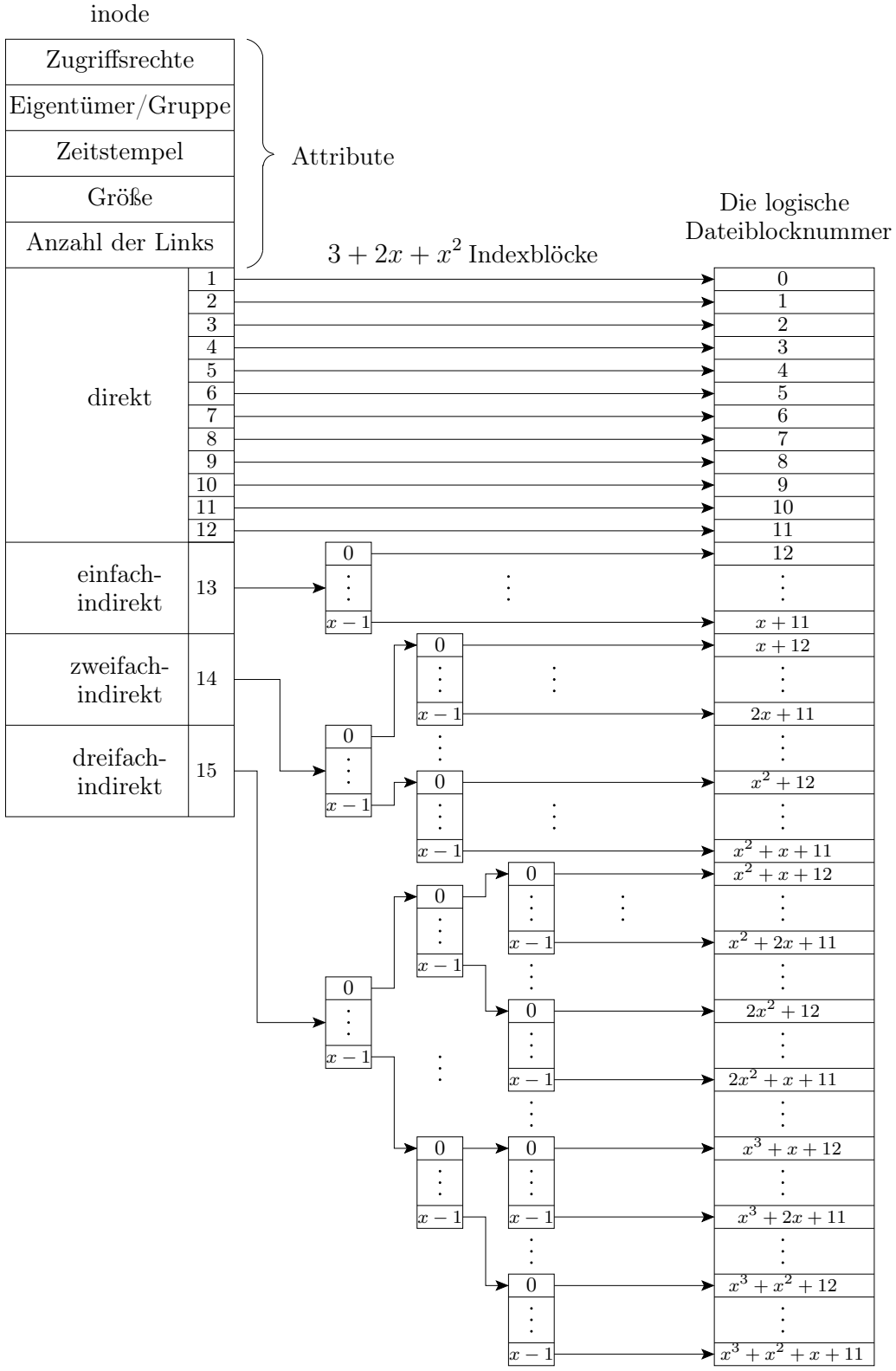

Abbildung 2.9: Ein Beispiel für einen *inode* mit Index- und Dateiblöcken, wobei jeder Indexblock x Blocknummern speichert.

Vorteile von inodes

Dieses Schema bietet mehrere Vorteile. Zum einen sind alle wesentlichen Informationen über eine Datei oder ein Verzeichnis im inode auf beschränktem Raum zusammengefasst; im Unterschied zur file-allocation table wird aber nur für die wirklich vorhandenen Objekte Speicherplatz belegt. Auf kurze Dateien — oder allgemeiner: auf die ersten 12 Blöcke jeder Datei — kann man sehr effizient zugreifen. Allgemein genügen maximal 5 Zugriffe auf den externen Speicher, um einen beliebigen Block einer langen Datei zu lesen. Wir sehen also: Durch inodes wird wahlfreier Zugriff recht gut unterstützt.

freie Blöcke

Die FAT bietet eine gute Übersicht darüber, welche Blöcke auf der Festplatte noch frei sind. Das leisten die inodes nicht. Man kann zu diesem Zweck zusätzlich einen langen *Bitvektor* verwenden, der für jeden Block ein Bit enthält, welches angibt, ob der Block frei oder belegt ist.

Der sequentielle Dateizugriff kann bei allen hier besprochenen Formen der nicht-zusammenhängenden Speicherung mühsam sein — wenn nämlich die Blöcke über die gesamte Magnetplatte verteilt sind. UNIX versucht, dieses Problem durch eine Anhebung der Blockgröße auf mehrere KByte zu mildern.

inode-Nummern

Die inodes der vorhandenen Dateien und Verzeichnisse sind in UNIX in einer Tabelle an fester Position auf der Festplatte gespeichert. In den Verzeichnissen (directories) steht beim Namen eines jeden enthaltenen Objekts die Nummer seines inodes in dieser Tabelle; man kann sich die inode-Nummern durch den zusätzlichen Parameter *i* des *ls*-Befehls ausgeben lassen.[26] Für jeden Prozess gibt es eine Liste mit den Namen aller Dateien, die dieser Prozess geöffnet[27] hat; diese Liste ist Teil des Prozesskontextes. Es kann durchaus vorkommen, dass mehrere Prozesse gleichzeitig auf dieselbe Datei zugreifen. Deshalb existiert zusätzlich ein systemweites Verzeichnis aller in Gebrauch befindlicher Dateien, auf das die Einträge in den Prozessverzeichnissen verweisen. In dem globalen Verzeichnis werden ebenfalls die Nummern der inodes verwendet.

Damit sind wir am Ende der ersten beiden Kurseinheiten über Betriebssysteme angekommen. Wir hoffen, die Lektüre hat Ihnen Vergnügen gemacht und Sie ein wenig dazu ermutigt, sich mit Linux zu beschäftigen — falls Sie nicht ohnehin schon mit Linux arbeiten. Für die Bearbeitung der nächsten beiden Kurseinheiten über Rechnernetze wünschen wir Ihnen viel Spaß und viel Erfolg!

[26] Der Befehl lautet dann also *ls -lsi*.

[27] Vergleiche hierzu Abschnitt 1.6.

Literatur

[1] A. Fiat, G. Woeginger, editors. *On-line Algorithms: The State of the Art*, volume 1442 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1998.

Literaturhinweise zu Betriebssystemen und speziell zu Linux/Unix siehe z. B. Seite 57 in Kurseinheit 1.

002684535
(10/21)

01801-6-02-S 1