DHBW Stuttgart

Frühjahr 2025

Betriebssysteme

Begleitbuch zur Vorlesung

Roland Weber

Inhaltsverzeichnis

1. Einstimmung

Sinn und Zweck eines Betriebssystems ist es, Programme laufen zu lassen. Das Betriebssystem stellt dazu Laufzeitumgebungen zur Verfügung, in denen die Programme Speicher vorfinden, Rechenzeit nutzen und die Möglichkeit zur Ein- und Ausgabe von Daten erhalten. Das beinhaltet Datei- und Netzwerkzugriffe sowie diverse Peripherie, die über Treiber zugänglich ist. Schnittstellen zum Zugriff auf solche Funktionen sind Teil der Laufzeitumgebungen. Rechner ab einer Smartwatch aufwärts sind in der Lage, mehrere Programme gleichzeitig auszuführen. Oder auch das gleiche Programm mehrfach, sofern das Sinn ergibt. Die Mechanismen, die das möglich machen, sind der Schwerpunkt meiner Betriebssysteme–Vorlesung und dieses Begleitbuchs. Für Rechner mit besonders einfacher Hardware oder besonders hohen Sicherheitsanforderungen gibt es aber weiterhin Betriebssysteme, die auf die Ausführung eines einzigen Programms ausgelegt sind.

1.1. Instanzen

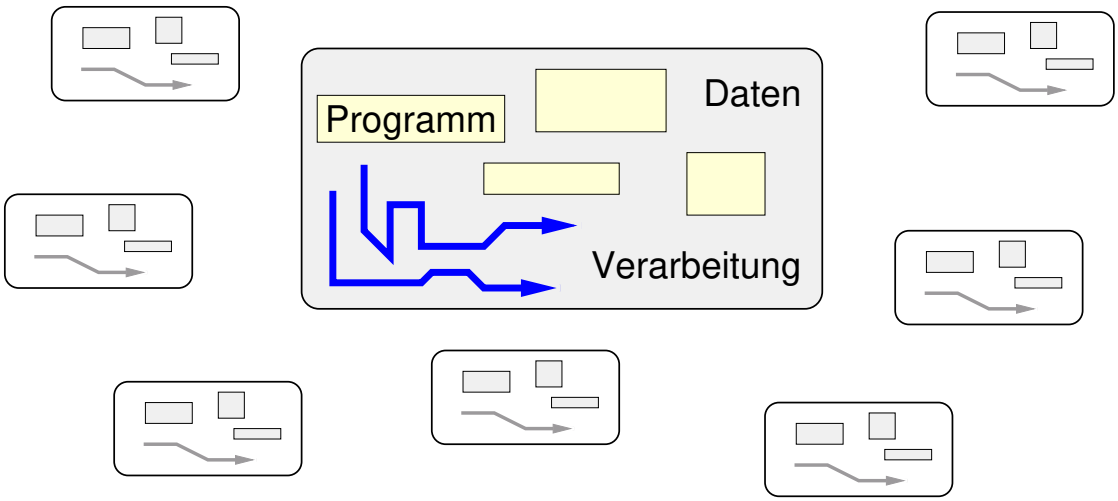


Abbildung 1.1.: Instanzen

Ein Programm ist eine Vorschrift zur Datenverarbeitung. Jedes laufende Programm bildet eine *Instanz*. Wie erwähnt kann auch das gleiche Programm in mehreren Instanzen laufen. Abbildung 1.1 zeigt schematisch viele Instanzen, von denen eine größer und mit mehr Details dargestellt ist. Die Rechtecke stehen für den Speicherinhalt der Instanz. Dort finden wir das Programm sowie die Daten, mit denen es arbeitet. Die verwinkelten Pfeile symbolisieren Befehlsströme, die den Programmcode ausführen. Befehlsströme brauchen Rechenzeit, um die Daten zu verarbeiten.
Abbildung 1.2 zeigt ebenfalls viele Instanzen, und zusätzlich einige *Systemfunktionen* in Form von Achtecken. Systemfunktionen stehen allen Instanzen zur Verfügung. Die Ab-

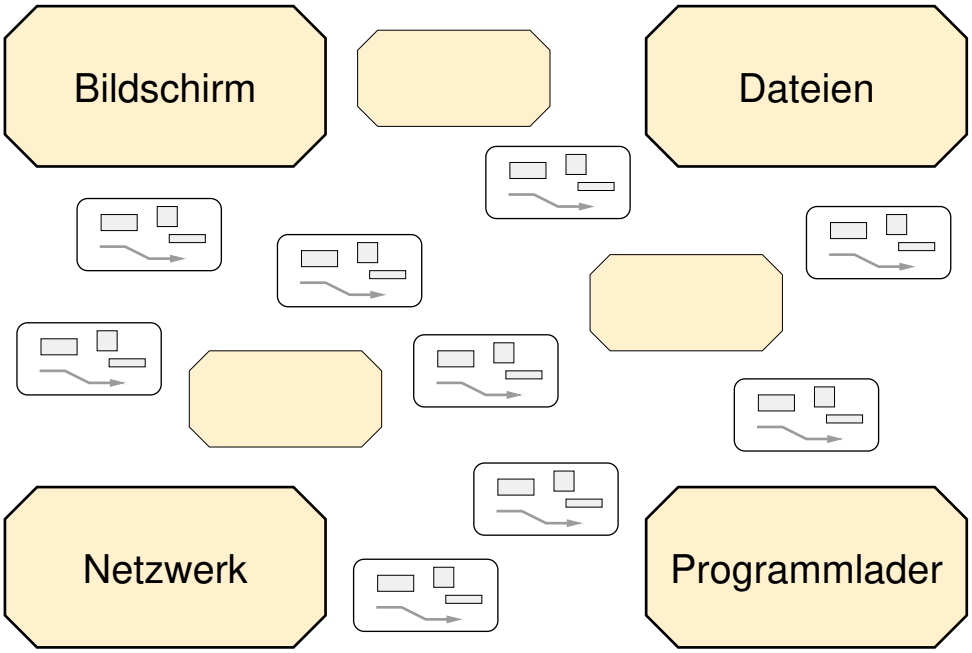


Abbildung 1.2.: Systemfunktionen

bildung benennt vier Beispiele. Bildschirm bzw. Grafikausgabe, eventuell auch mit Touch zur Eingabe, steht für Peripheriegeräte im allgemeinen. Dateien repräsentieren den Zugriff auf Massenspeicher, Netzwerk die Kommunikation mit anderen Rechnern oder Geräten. Der Programmlader ist eine Funktion, die Instanzen erzeugt. Dieses letzte Beispiel zeigt, dass Funktionen nicht nur Hardware ansteuern, sondern auch reine Software sein können. Allerdings greift der Programmlader auf Dateien zu, die eventuell auf einem Netzlaufwerk liegen. Systemfunktionen bauen also bei Bedarf aufeinander auf.
In Abbildung 1.3 werden weitere Details von Instanzen erkennbar. Neben dem Programm und zu verarbeitenden Daten enthalten sie meist auch *Bibliotheken*. Eine Bibliothek enthält wiederverwendbaren Code, auf den verschiedene Programme und Instanzen zurückgreifen können. In der links oben eingezeichneten Instanz dienen zwei der drei Bibliotheken dem Zugriff auf zwei Systemfunktionen. Andere Instanzen, die die gleichen Systemfunktionen verwenden, nutzen dazu ebenfalls die gleichen Bibliotheken. Die Bibliotheken enthalten die Programmierschnittstellen der Systemfunktionen.

Ein wichtiges Prinzip der Softwaretechnik ist die Wiederverwendung von Code. Das geschieht auf vielen Ebenen und Granularitätsstufen. Programmiersprachen bieten die Möglichkeit, Code in Funktionen oder Klassen zu kapseln, die in Namensräumen zu Paketen kombiniert werden. Auf dieser Ebene sind Compiler, Linker und Interpreter die Werkzeuge, die Wiederverwendung ermöglichen.
Auf Ebene der Betriebssysteme bestehen die wiederverwendbaren Module aus Programmen und dynamisch gebundenen Bibliotheken. Diese liegen in separaten Dateien und werden vom Programmlader zu Instanzen kombiniert. Die nächsthöhere Ebene bilden Container, welche Instanzen und Systemfunktionen kapseln, und die selbst zu Pods[1] kombiniert

[1] `https://kubernetes.io/docs/concepts/workloads/pods/`

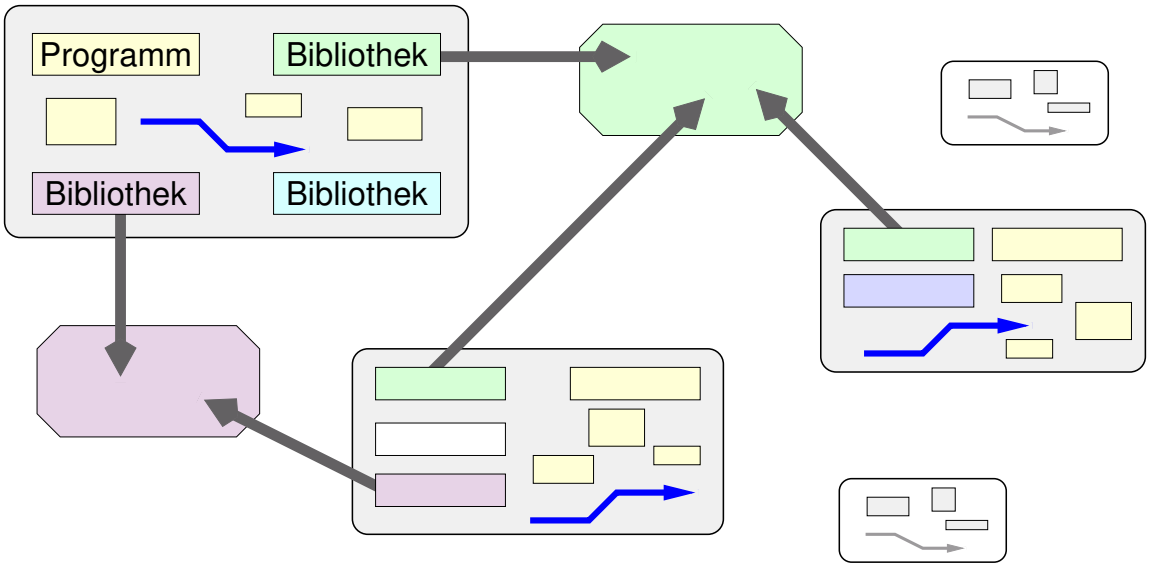


Abbildung 1.3.: Zugriff auf Systemfunktionen

werden. Darüber liegen verteilte Systeme aus mehreren vernetzten Rechnern, in denen Instanzen, Pods oder virtuelle Maschinen die Module bilden.

In den Abbildungen sind Instanzen als Rechtecke mit abgerundeten Ecken dargestellt. Der Code und die Daten, auf die eine Instanz unmittelbar zugreifen kann, liegen innerhalb dieses Bereichs. Jede Instanz ist somit in sich abgeschlossen und von den anderen getrennt. Das Betriebssystem sollte dafür sorgen, dass Instanzen sich nicht gegenseitig stören, sei es durch Programmierfehler oder böse Absicht.
Gleichzeitig sollen Instanzen aber auch miteinander arbeiten dürfen. Dabei kommt den Systemfunktionen eine tragende Rolle zu. Sie liegen außerhalb der Instanzen und lassen sich trotzdem von diesen nutzen. Mehrere Instanzen, die die gleiche Systemfunktion verwenden, können dort Daten hinterlegen oder abholen. Das muss nicht bedeuten, dass eine Instanz in eine Datei schreibt, welche eine andere Instanz liest. Manche Systemfunktionen etablieren direkte Kommunikationswege zwischen Instanzen, oder erstellen sogar gemeinsam genutzte Speicherbereiche.

1.2. Infrastruktur

Die bisherigen Abbildungen in Abschnitt 1.1 sind als Draufsicht zu verstehen. Ob Instanzen im Norden oder Süden einer Abbildung liegen, spielt dort keine Rolle. Das gleiche gilt für die Systemfunktionen. Mit Abbildung 1.4 wechselt die Perspektive zu einer Seitenansicht. Es gibt oben und unten, durch eine deutliche Grenze getrennt. Oben liegen Instanzen. Unten befindet sich die Infrastruktur, der Kern des Betriebssystems. Ich verwende die abstrakten, deutschen Begriffe Instanzenbereich und Infrastruktur aus der Wettstein'schen Systemarchitektur.[1] Gebräuchliche englische Begriffe dafür sind *userland* und *kernel space*. Diese haben einen technischen Bezug, der später klar wird.

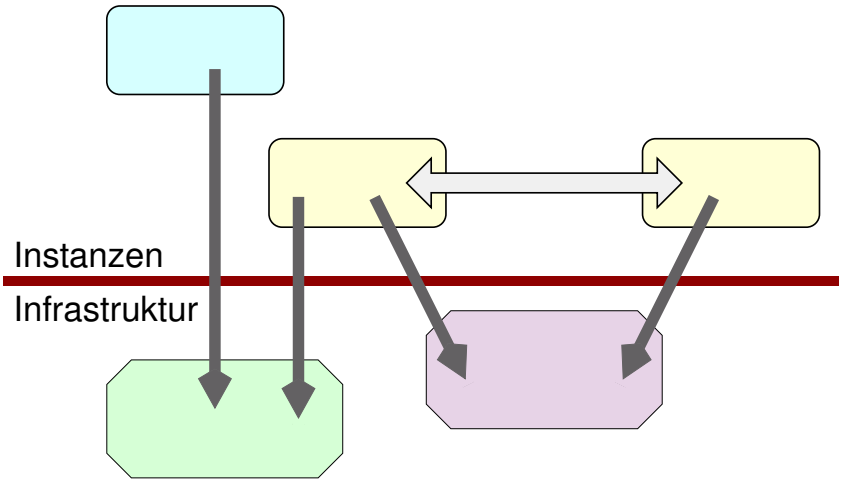


Abbildung 1.4.: Instanzen und Infrastruktur

Wie erwähnt sind Instanzen in sich abgeschlossen und arbeiten jeweils mit ihren eigenen Daten. Die Infrastruktur setzt diese Kapselung durch, sofern die technischen Voraussetzungen dafür vorliegen. Der einzig gültige Weg aus einer Instanz heraus sind direkte Aufrufe der Infrastruktur und der dort bereitgestellten Systemfunktionen. In Abbildung 1.4 ist das durch die dunkelgrauen Pfeile von oben nach unten dargestellt.
Zwei der Instanzen sind durch einen hellgrauen Pfeil mit Doppelspitze verbunden. Das symbolisiert eine logische Beziehung, diese beiden Instanzen arbeiten miteinander. Das geschieht aber nicht, indem eine Instanz in die andere hineinreicht. Technisch umgesetzt wird die logische Beziehung, indem beide Instanzen eine gemeinsame Systemfunktion aufrufen, die zwischen ihnen vermittelt. Da alle Aufrufe aus einer Instanz heraus in die Infrastruktur führen, kann die Infrastruktur auch kontrollieren, ob Aufrufe gültig sind und die aufrufende Instanz über ausreichende Berechtigungen verfügt.

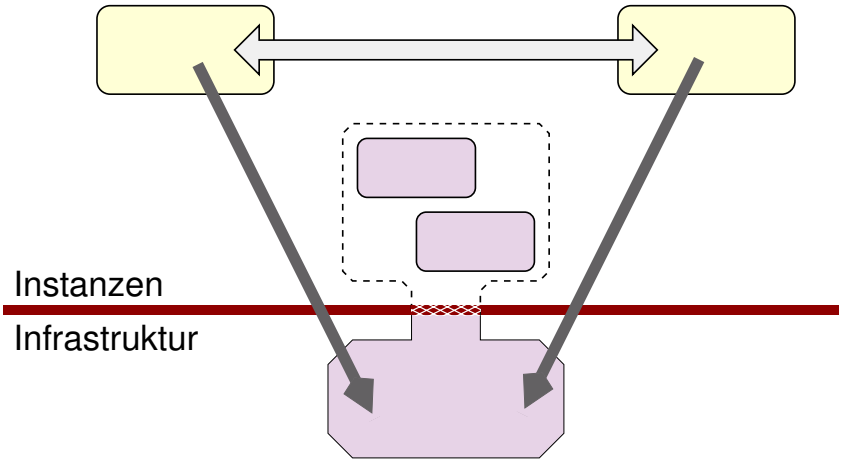


Abbildung 1.5.: Systemfunktion mit Instanzen

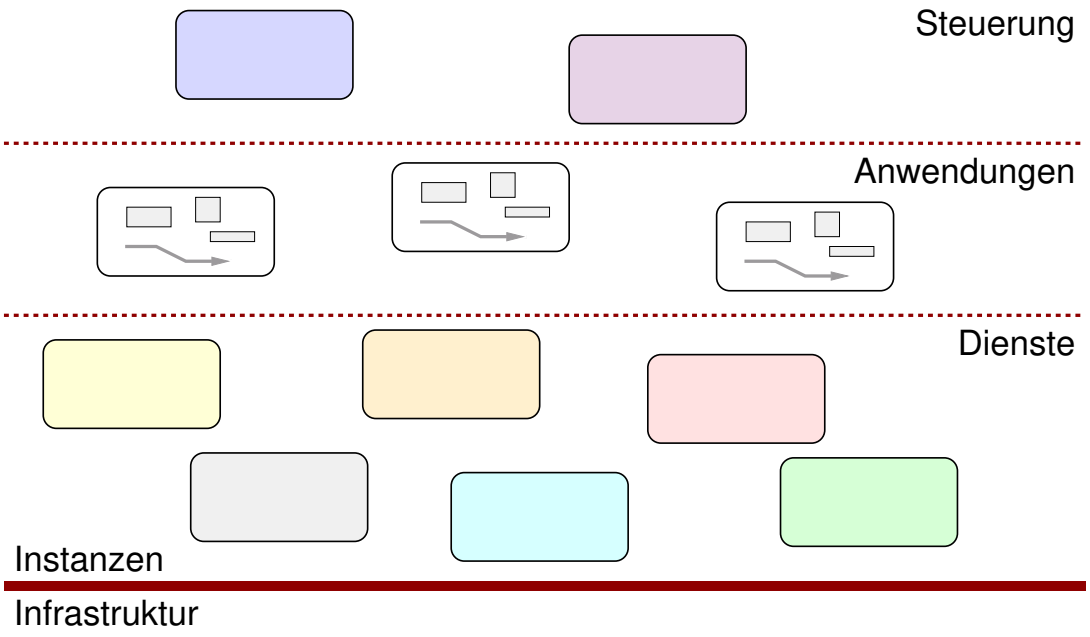
Systemfunktionen sind nicht auf die Infrastruktur beschränkt. Wenn ein Betriebssystem schon die Möglichkeit bietet, mehrere Instanzen gleichzeitig auszuführen, nutzt man das

auch zur Strukturierung der Funktionen des Betriebssystems selbst. Abbildung 1.5 greift noch einmal das Beispiel der zwei miteinander arbeitenden Instanzen auf. Beide verwenden die gleiche Systemfunktion, durch Aufrufe der Infrastruktur. Ob diese Funktion tatsächlich in der Infrastruktur angesiedelt ist, oder ob die Aufrufe erst an andere Instanzen weitergeleitet werden, bevor sich ihre Wirkung entfaltet, spielt für die Aufrufer keine Rolle. Der Zugang zu Systemfunktionen erfolgt über die Infrastruktur. Die Implementierung einer Systemfunktion kann in der Infrastruktur liegen, im Instanzenbereich, oder über beides hinweg aufgeteilt sein.
Nicht nur das Betriebssystem bietet Funktionen an. Auch Anwendungen können einen Teil ihrer Aufgaben in Form von Funktionen implementieren. Bei sogenannter *Middleware* wie Datenbanken oder Message–Queueing–Systemen ist das Bereitstellen einer Funktion sogar der Hauptzweck.

1.3. Wettstein'sches Schichtenmodell

Wenn Instanzen miteinander in Beziehung stehen, gibt es auch Abhängigkeiten zwischen ihnen. Bestimmte Instanzen funktionieren nur dann, wenn bestimmte andere Instanzen vorhanden sind und funktionieren. Das kann zu Problemen führen, wenn durch ungeplante Abhängigkeitsketten zum Beispiel zyklische Abhängigkeiten entstehen. Um den Überblick zu behalten, bietet es sich an, die Instanzen zu ordnen. Dazu dient das Wettstein'sche Schichtenmodell.[1]
Abbildung 1.6 zeigt viele Instanzen, gruppiert in drei übereinander liegende Schichten. Die *Anwendungen* stehen in der mittleren der drei Schichten. Darunter liegen *Dienste*, darüber steht eine *Steuerung*. Die Grundidee dabei ist, dass Instanzen einer Schicht auf diejenigen

Abbildung 1.6.: Wettstein'sches Schichtenmodell



aller darunter liegenden Schichten zugreifen dürfen, ohne dass zyklische Abhängigkeiten entstehen. Die Abhängigkeiten innerhalb einer Schicht sollten sorgfältig geplant werden. In Ausnahmefällen ist es auch notwendig, von einer tieferen Schicht auf eine höhere zuzugreifen. Dann ist besondere Aufmerksamkeit geboten, um die Stabilität des Gesamtsystems nicht zu gefährden.
Die Infrastruktur bildet eine vierte Schicht. Wie weiter oben besprochen sind alle Instanzen von der Infrastruktur abhängig. Beziehungen zwischen Instanzen werden durch die Infrastruktur erst möglich. Der Unterschied zwischen Infrastruktur und Instanzen ist technischer Natur. Die Gruppierung in Instanzenschichten dagegen ist eine logische oder vielmehr menschliche Unterscheidung. Technisch funktionieren alle Instanzen gleich. Deshalb ist die Trennlinie zwischen Instanzen und Infrastruktur klar ausgeprägt, während die Grenzen zwischen den Instanzenschichten nur angedeutet erscheinen.
Das Wettstein'sche Schichtenmodell existiert in mehreren Detailgraden. In der gröbsten Stufe unterscheidet es nur zwischen Instanzen und Infrastruktur. Abbildung 1.6 zeigt das einfache Modell mit drei Instanzenschichten und der Infrastruktur. Die feinste Unterteilung in sieben Instanzen- und drei Infrastrukturschichten bespreche ich nicht mehr. Innerhalb einer Instanz oder einer Anwendung kann man wiederum das Wettstein'sche Schichtenmodell anwenden, um zum Beispiel verschiedene Module oder Teilaufgaben zu ordnen.

1.4. Knackpunkte

- Ein Betriebssystem stellt Laufzeitumgebungen für Programme bereit.
- Eine Instanz ist ein laufendes Programm.
- Mehrere Instanzen laufen gleichzeitig, voneinander getrennt.
- Instanzen können Systemfunktionen nutzen.
- Instanzen arbeiten zusammen, indem sie gemeinsame Systemfunktionen nutzen.
- Die Infrastruktur, d.h. der Kern des Betriebssystems, kapselt Instanzen und erlaubt ihnen den Aufruf von Systemfunktionen.
- Anwendungen, Middleware und das Betriebssystem selbst sind als Instanzen und Systemfunktionen strukturiert.
- Das Wettstein'sche Schichtenmodell ordnet Instanzen, Systemfunktionen und mehr.
  - Über den Anwendungen gibt es eine Steuerung.
  - Dienste liegen unter den Anwendungen und der Steuerung.
- Programme und dynamisch gebundene Bibliotheken sind Software–Module.
- Ein Programmlader erstellt Instanzen aus einem Programm und dynamisch gebundenen Bibliotheken.

2. Ausgewählte Interaktionsmuster

In einem Betriebssystem, welches mehrere Programminstanzen und Befehlsströme gleichzeitig ausführen kann, bilden diese Instanzen und Befehlsströme eine Möglichkeit zur Strukturierung. Sie werden zu Modulen, die mittels *Interaktion* zusammenarbeiten und potentiell wiederverwendbar sind. Mechanismen zur Interaktion, und die Implementierung solcher Mechanismen, sind ein zentrales Thema meiner Vorlesung und späterer Kapitel. In diesem Kapitel stelle ich einige Interaktions*muster* vor, mit denen man die Zusammenarbeit von Befehlsströmen strukturieren kann. Sie lassen sich flexibel miteinander kombinieren. Zur Darstellung dienen Interaktions*diagramme*.
Es gibt sehr viele gängige Interaktionsmuster. Ich beschränke mich auf eine kleine Auswahl, da es hier in erster Linie darum geht, einige nützliche und anschauliche Beispiele für strukturierte Interaktion kennenzulernen. Zudem haben die verwendeten Interaktionsdiagramme nur eng begrenzte Möglichkeiten, Muster verständlich zu machen. Die Auswahl orientiert sich an Professor Horst Wettstein's Buch *Systemarchitektur*.[1] Für das Verständnis späterer Kapitel ist es nicht notwendig, alle vorgestellten Interaktionsmuster zu kennen. Es genügen ein oder zwei.
Zunächst erklärt Abschnitt 2.1 die wichtigsten Elemente in Interaktionsdiagrammen. Die Abschnitte 2.2 und 2.3 präsentieren zwei klassische Interaktionsmuster zwischen zwei Befehlsströmen bzw. Prozessen: Client/Server und Erzeuger/Verbraucher. Die Reproduktion in Abschnitt 2.4 bringt Datenparallelität ins Spiel. Danach folgen mit dem Fließband (2.5), der Pufferung (2.6) und dem Team (2.7) weitere Muster zum Kombinieren. Abschnitt 2.8 fasst die wichtigsten Punkte dieses Kapitels zusammen.

In allen folgenden Beispielen kommunizieren die Befehlsströme mittels Nachrichten. Eine andere Art der Interaktion, nämlich die *Synchronisation* ohne Datenübertragung, ist hier nicht berücksichtigt. Viele Beispiele dafür enthält das *Little Book of Semaphores* [2] in Form von Aufgaben mit Lösungen. Die Beschäftigung mit solchen Aufgaben hilft dabei, das Gehirn in die richtigen Windungen zu legen, um später Synchronisations- oder Protokollfehler zu vermeiden. Diese sind oft schwer zu entdecken, da sie von den relativen Geschwindigkeiten der Befehlsströme abhängen (engl.: *race conditions*) und sich deshalb nicht verlässlich reproduzieren lassen.

2.1. Darstellung

Abbildung 2.1 zeigt ein Interaktionsdiagramm mit zwei Befehlsströmen, die Nachrichten über einen Kanal austauschen. Befehlsströme sind als breite, blaue Pfeile eingezeichnet. Sie kommen in Pfeilrichtung voran. Der Kanal liegt als gelber, zweigeteilter Kreis in der Mitte. Er unterstützt die Operationen Senden (S) und Empfangen (E). Dünne, schwarze Pfeile markieren, wo die Befehlsströme die jeweilige Operation aufrufen. Der Kanal kann Nachrichten zwischenspeichern. Deshalb ist das Senden asynchron, der aufrufende Befehlsstrom arbeitet sofort weiter. Das Empfangen dagegen ist synchron. Der Befehlsstrom muss vielleicht warten, bis eine Nachricht am Kanal eintrifft. Das zeigt der schwarze Querbalken an der Aufrufstelle an. Der Befehlsstrom rechts läuft in einer Schleife, könnte also prinzipiell mehrere Nachrichten hintereinander empfangen. Allerdings sendet der Befehlsstrom links nur eine Nachricht.

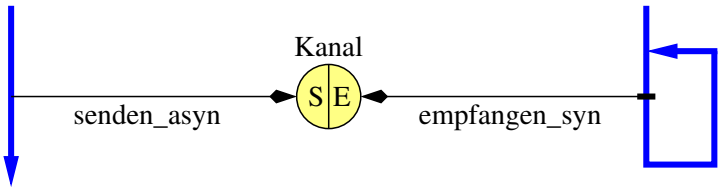


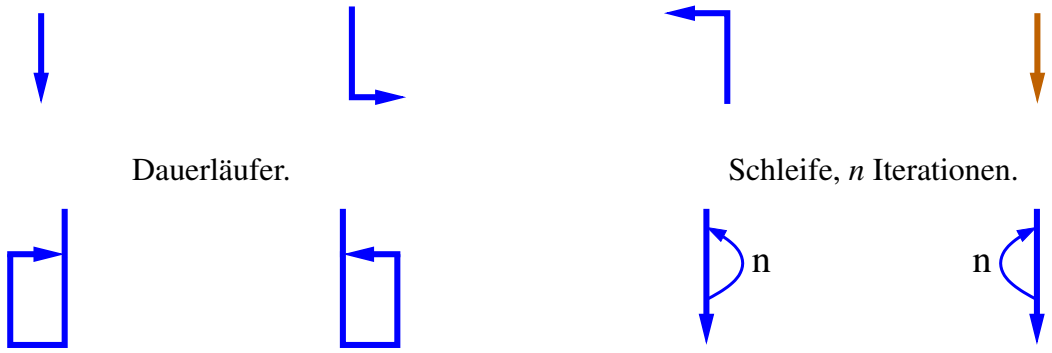
Abbildung 2.1.: Darstellung als Interaktionsdiagramm

Der Begriff „Befehlsstrom“ ist sehr hardwarenah. Er betont, wie ein Prozessor arbeitet. In den Interaktionsdiagrammen steht aber die Software im Vordergrund. Deshalb wechsle ich nun zum abstrakteren Begriff „Prozess“. Die Unterschiede liegen in Details, die Kapitel 8 beleuchtet. Prozesse in Interaktionsdiagrammen übernehmen bestimmte Rollen. Aus dieser Perspektive zeigt Abbildung 2.1 links einen Prozess in der Rolle des Senders, rechts einen Empfänger. Instanzen tauchen in Interaktionsdiagrammen nicht auf. Die Muster funktionieren mit Prozessen in verschiedenen Instanzen ebenso wie mit Prozessen in der gleichen Instanz.
Die hier vorausgesetzten Kanäle, welche Kapitel 11 ausführlich beschreibt, sind bewusst sehr einfach gehalten. Ein Empfänger bekommt die nächstbeste verfügbare Nachricht. Jede gesendete Nachricht wird dem nächstbesten verfügbaren Empfänger zugestellt. Für unterschiedliche Adressaten braucht man getrennte Kanäle. Viele Betriebssysteme bieten auch höherwertige Kommunikationsmechanismen an. Zum Beispiel Kanäle, aus denen sich mehrere Empfänger bestimmte Nachrichten herauspicken können. Oder Verbindungen, die zwischen zwei bestimmten Prozessen aufgebaut werden und den Datenaustausch in beide Richtungen erlauben. Interaktionsmuster, die mit den einfachen Kanälen funktionieren, kann man im Allgemeinen auch mit höheren Mechanismen gut implementieren.[1] Umgekehrt können höhere Mechanismen Interaktionsmuster unterstützen, die sich mit den einfachen Kanälen nur umständlich implementieren lassen.

Abbildung 2.2 zeigt eine Übersicht der wichtigsten Elemente, die in den Interaktionsdiagrammen dieses Kapitels auftauchen. In Kapitel 9 kommen neben den Kanälen weitere Interaktionsobjekte ins Spiel.

[1] Ausnahmen bestätigen die Regel

Befehlsströme. Kommen in Pfeilrichtung voran. Meist blau.



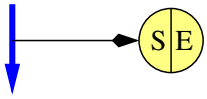
Dauerläufer.

Schleife, *n* Iterationen.

Kanäle zur Interaktion. Zwei Operationen, Senden und Empfangen. Meist gelb.

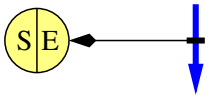
Aufrufe von Befehlsströmen an Kanälen.

Senden asynchron.



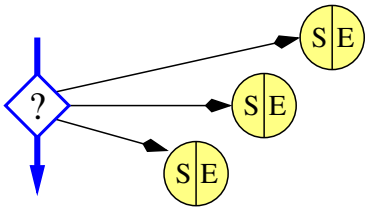
Befehlsstrom läuft weiter.

Empfangen synchron.



Befehlsstrom wartet auf Nachricht.

Senden an *einen* von mehreren Kanälen.
Befehlsstrom entscheidet an welchen.



Verzweigung im Befehlsstrom.

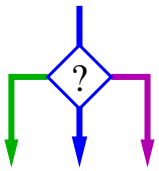


Abbildung 2.2.: Elemente der Interaktionsdiagramme

2.2. Client/Server

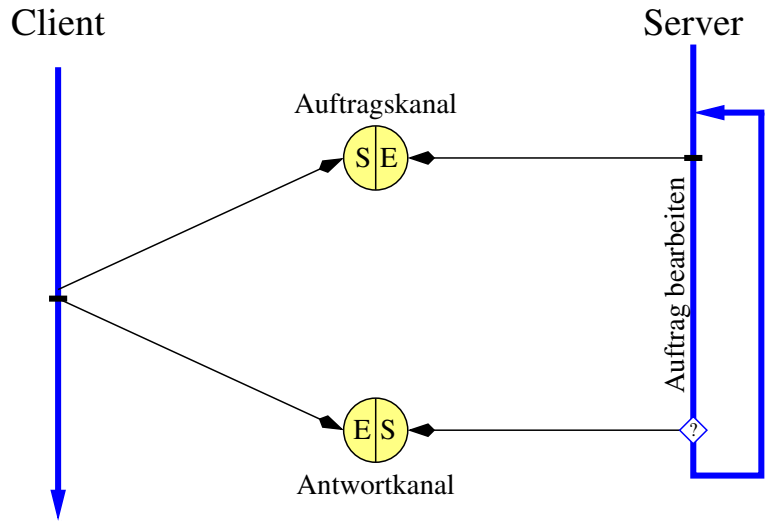


Abbildung 2.3.: Client und Server

Beispiele: Webserver, Datenbankserver, *File Server*, *Remote Procedure Call*, *Language Server* `https://langserver.org`

Client/Server ist das einfachste Interaktionsmuster zur Beauftragung und das wohl verbreitetste überhaupt. Ein Server bietet bestimmte Dienste an. Clients schicken einen Auftrag und erhalten nach Erledigung eine Antwort. Abbildung 2.3 zeigt einen Client, einen Server und zwei Kanäle für Aufträge bzw. Antworten. Gleich nach dem Senden des Auftrags wartet der Client hier auf die Antwort. Das Verhalten ähnelt einem lokalen Funktionsaufruf. Vor dem Aufruf müssen alle Parameter bekannt sein, bei der Rückkehr liegt das Ergebnis vor.

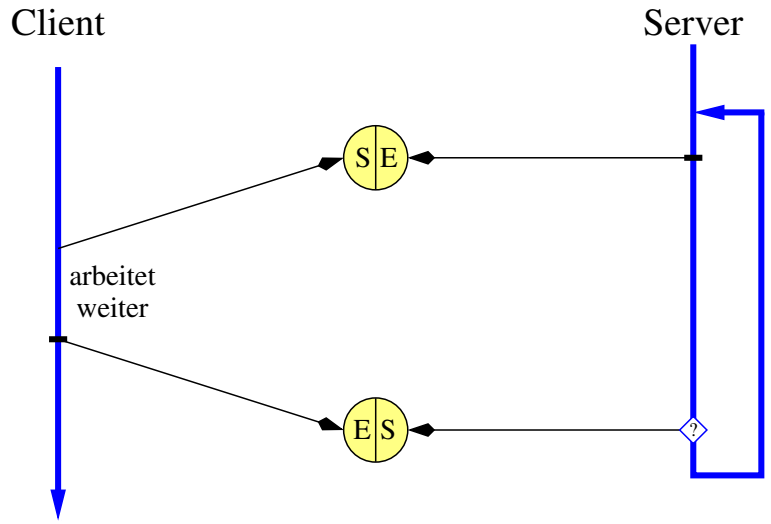


Abbildung 2.4.: Asynchroner Auftrag

Eine Variante zeigt Abbildung 2.4. Hier arbeitet der Client nach dem Senden des Auftrags noch weiter und wartet erst später auf die Antwort. Er setzt also einen *asynchronen Auftrag* ab. Das ermöglicht Parallelität zwischen Client und Server. Voraussetzung ist, dass die Parameter für den Aufruf rechtzeitig vorher bekannt sind und der Client noch etwas Sinnvolles erledigen kann, ohne das Ergebnis zu kennen. Im besten Fall liegt dann das Ergebnis schon vor, wenn der Client seine Empfangsoperation erreicht.

2.2.1. Viele Clients

Der einzige Auftrag vom einzigen Client in den ersten beiden Abbildungen steht stellvertretend für viele Aufträge, die von einem oder mehreren Clients stammen können. Zur Verdeutlichung zeigt Abbildung 2.5 drei Clients. Hier wird der Bezug von Kanälen zu Prozessen klar. Der Auftragskanal gehört zum Server, es gibt nur einen. Aufgrund des sehr einfachen Kanalmodells, das den Interaktionsdiagrammen zugrunde liegt, braucht aber jeder Client einen eigenen Antwortkanal. Sonst könnten die Antworten durcheinander geraten und beim falschen Client landen.

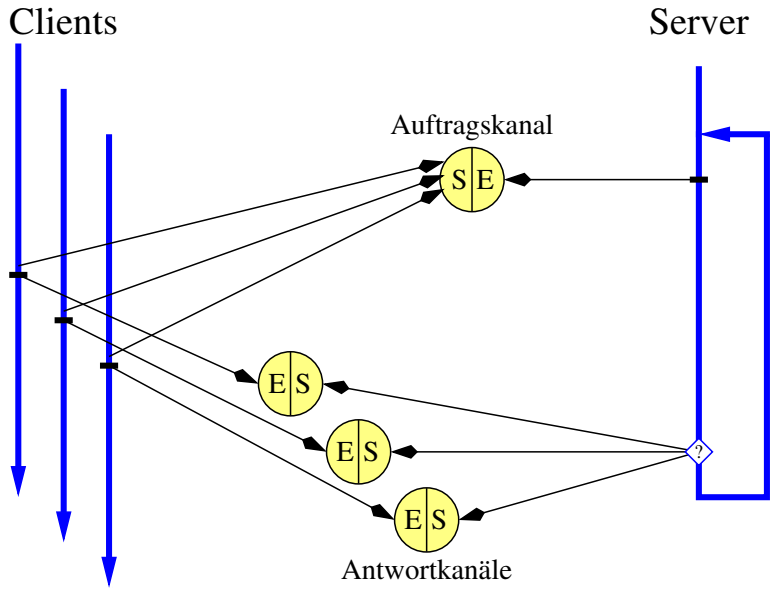


Abbildung 2.5.: Mehrere Clients und ein Server

Die Sendeoperation des Servers ist mit einem Fragezeichen in einer Raute dargestellt. Das bedeutet, dass der Server an dieser Stelle entscheidet, an welchen der möglichen Kanäle er die Antwort schickt. Trotz mehrerer Pfeile in der Abbildung sendet er also in jedem Durchgang nur eine Nachricht an einen der Kanäle. In den Abbildungen zuvor ist der Sinn dieser Raute nicht erkennbar, da ohnehin nur ein Kanal zur Auswahl steht.
Falls der tatsächlich verwendete Kommunikationsmechanismus keine Zuordnung erlaubt, muss der Client mit in den Auftrag schreiben, an welchen Kanal die Antwort gehen soll. In der Praxis kommen oft bidirektionale Verbindungen zum Einsatz. Dann schickt der Server die Antwort über die gleiche Verbindung, über die der Auftrag eintraf. Das Aufbauen der Verbindung kann man als Erzeugen eines Antwortkanals für den Client interpretieren.

2.2.2. Mehrere Server

Ein einzelner Server–Prozess kann Aufträge nur sequentiell bearbeiten. Bei vielen Aufträgen von vielen Clients, wie gerade in Abschnitt 2.2.1 beschrieben, gerät ein einzelner Server schnell an die Grenzen der Hardware. Selbst wenn er die Auftragslast im Mittel noch bewältigen kann, steigt die Wartezeit für Clients.
In der Praxis setzt man gerne mehrere Server–Prozesse ein, wie in Abbildung 2.6 dargestellt. Jeder Client sendet seinen Auftrag weiterhin an den gleichen Kanal. Dahinter warten aber mehrere Prozesse. Jeder Auftrag wird von genau einem dieser Prozesse empfangen und bearbeitet. In allen Prozessen läuft die gleiche Software, die gleiche Logik. Es spielt also keine Rolle, welcher Prozess einen Auftrag übernimmt.

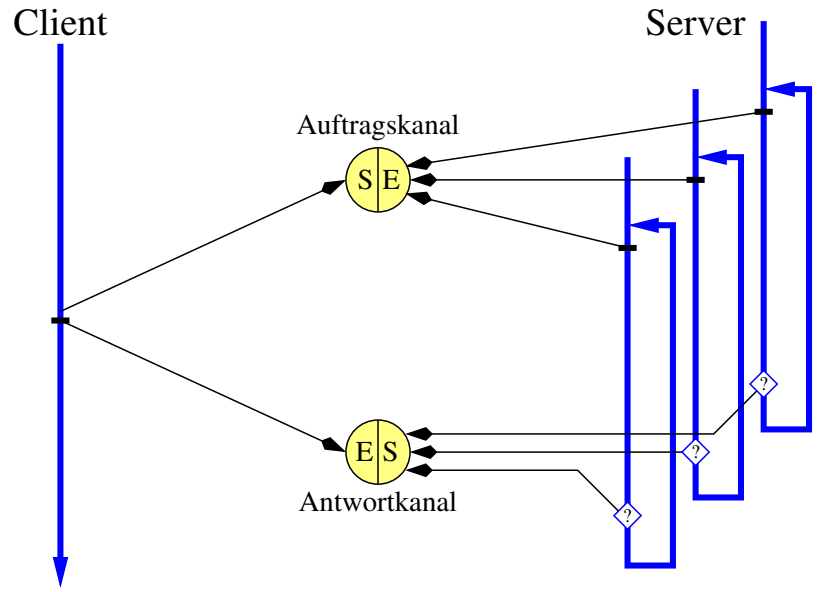


Abbildung 2.6.: Reproduzierter Server

Dies ist ein erstes Beispiel für das Interaktionsmuster *Reproduktion*, in diesem Fall ein reproduzierter Server. Auch in den noch folgenden Interaktionsmustern werden wir der Reproduktion immer wieder begegnen. Charakteristisch ist jedes Mal, dass Arbeit an einem bestimmten Punkt, hier dem Auftragskanal, anfällt. Diese Arbeit teilen sich mehrere Prozesse, die alle die gleiche Software ausführen. Mehrere Clients sind keine Reproduktion, selbst wenn sie die gleiche Software ausführen. Clients empfangen keine Arbeit an einem gemeinsamen Kanal.

2.2.3. Verschachtelte Aufträge

In der Einleitung dieses Kapitels habe ich bereits erwähnt, dass sich Interaktionsmuster miteinander kombinieren lassen. Das klappt auch schon mit Client/Server als dem einzigen Muster. Ein einzelner Prozess kann mehrere Rollen übernehmen und sowohl Server als auch Client sein. Abbildung 2.7 zeigt verschachtelte Client/Server–Beziehungen mit synchronen und asynchronen Aufträgen. Zur Orientierung habe ich Antwortkanäle hier grün statt gelb gefärbt. Die drei Prozesse links sind reine Clients, die beiden Prozesse rechts reine Server. Die beiden Prozesse in der Mitte treten in beiden Rollen auf, als Server für die linken Prozesse und als Client für die rechten.

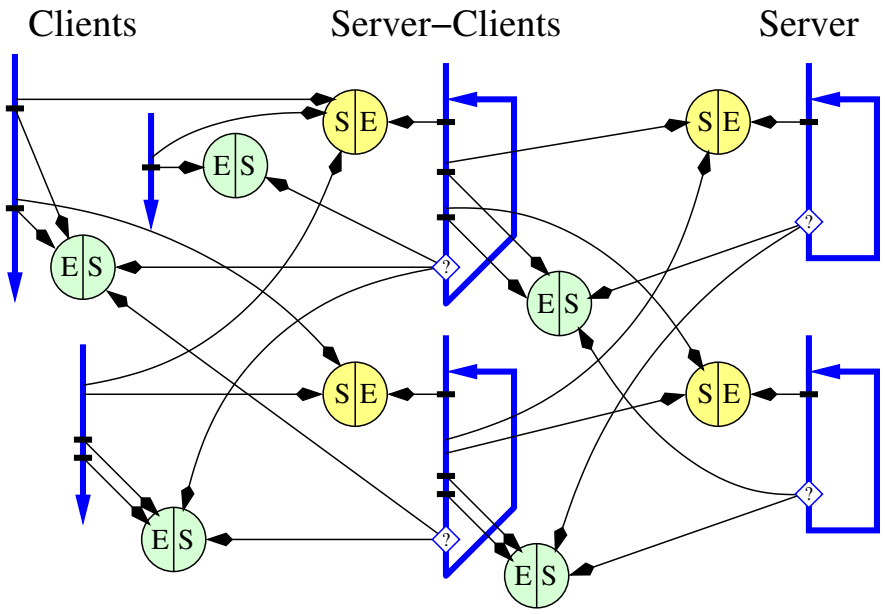


Abbildung 2.7.: Verschachtelte Client/Server–Beziehungen

Der Client links oben schickt zwei Aufträge synchron, erst zum oberen Server in der Mitte, dann zum unteren. Gleich rechts davon ist ein Client, der nur einen Auftrag schickt. Der dritte Client links unten schickt zwei Aufträge asynchron und überlappend. Dadurch kann er gleich zwei Server parallel für sich einspannen. In der Praxis arbeiten zum Beispiel Webbrowser auf diese Weise, wenn eine Webseite Bilder, Stylesheets oder Skripte von verschiedenen Webservern einbindet.
Die beiden Prozesse in der Mitte setzen pro Auftrag, den sie bearbeiten, zwei weitere Aufträge ab. Der obere Prozess schickt die Aufträge synchron, der untere asynchron. Ihre Arbeitsweise ähnelt also den beiden Clients ganz links. Die Aktionen sind aber in die Server–Hauptschleife eingebunden und werden nur im Rahmen der Bearbeitung eines Auftrags ausgeführt.

Hinweis: *Im Folgenden verzichte ich auf solche wild zusammengesetzten Beispiele. Für die Vorlesung gebe ich Aufgaben zur Kombination von Interaktionsmustern aus. Mit Musterlösungen, wie es sich gehört.*

2.3. Erzeuger/Verbraucher

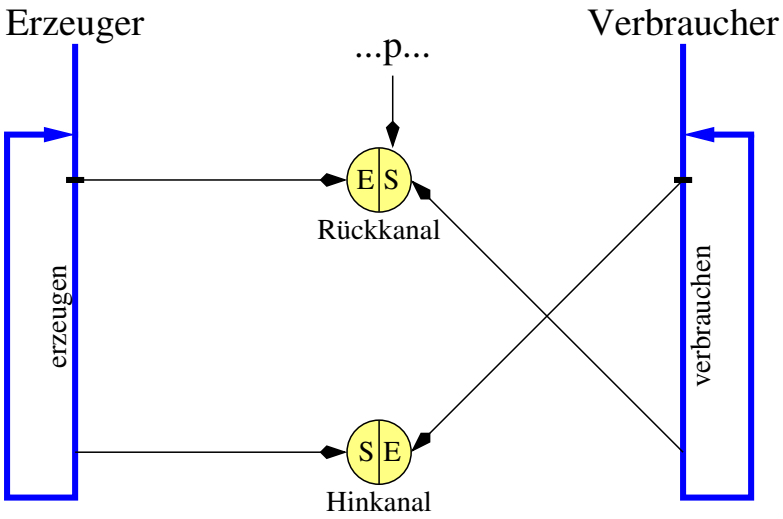


Abbildung 2.8.: Erzeuger und Verbraucher

Beispiele: Immer dann, wenn begrenzter Puffer von einem Prozess gefüllt und von einem anderen geleert wird. Vielleicht generiert ein Prozess Daten und schreibt sie in einen Strom. Der andere schickt sie weiter über eine Netzwerkverbindung oder speichert sie auf eine Festplatte. Je nach Gestaltung kann auch ein einzelner Kanal mit seinen Sendern und Empfängern ein Erzeuger/Verbraucher–System bilden.

Betrachten wir zunächst zwei Prozesse, die nicht zu einem Erzeuger/Verbraucher–System gekoppelt sind. Abbildung 2.9 zeigt links eine Quelle, rechts eine Senke für Daten. Dazwischen liegt ein Kanal, der Nachrichten bis zum Empfang zwischenspeichert, also Daten puffert. Die Quelle erzeugt und sendet Daten, so schnell sie kann, die Senke empfängt und verarbeitet diese Daten, ebenfalls so schnell sie kann.
Ein Problem gibt es, wenn die Quelle vorübergehend oder dauerhaft schneller arbeitet, als die Senke die Daten verarbeiten kann. Der Kanal bräuchte dann immer mehr Pufferspeicher, ohne dass man eine Obergrenze angeben könnte. Das funktioniert in der Praxis natürlich nicht. Sollen keine Daten verloren gehen, muss man die Quelle bremsen, damit die Senke mit der Verarbeitung nachkommt.

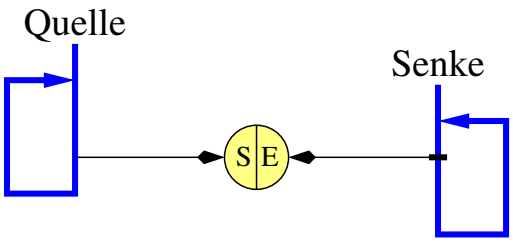


Abbildung 2.9.: Unbeschränkte Datenquelle

Man braucht also Mechanismen zur Datenflusssteuerung. Bei seriellen Schnittstellen gibt es dazu unter anderem die Steuerzeichen XON und XOFF. Das Netzwerkprotokoll TCP dagegen definiert ein Empfangsfenster (engl.: *receive window*) beschränkter Größe. Der Sender darf höchstens so viele Daten losschicken, wie in das Empfangsfenster passen. Dann muss er auf Bestätigungen vom Empfänger warten. Dieser Ansatz ist dem hier vorgestellten Erzeuger/Verbraucher–System schon sehr ähnlich.
Abbildung 2.8 auf der vorhergehenden Seite zeigt ein Erzeuger/Verbraucher–System, das auf zwei Kanälen beruht. Der Erzeuger, also die Quelle, empfängt zuerst eine Nachricht am Rückkanal, bevor er Daten generiert. Diese Nachricht entspricht der Erlaubnis, eine Nachricht voll Daten zu erzeugen und am Hinkanal zu senden. Der Verbraucher, also die Senke, empfängt Nachrichten vom Hinkanal, verarbeitet die enthaltenen Daten, und bestätigt dies anschließend mit einer Nachricht an den Rückkanal.
Damit dieses Wechselspiel in Gang kommt, muss zu Beginn mindestens eine Nachricht im Rückkanal liegen, so dass der Erzeuger loslegen kann. Sinnvollerweise initialisiert man das System mit mindestens zwei Nachrichten, damit Erzeuger und Verbraucher auch parallel arbeiten können. Mehr initiale Nachrichten können Wartezeiten auf beiden Seiten verringern, vergrößern aber auch den Speicherbedarf, In Abbildung 2.8 sind es p Nachrichten, also eine konfigurierbare Anzahl, die zu Beginn aus dem Nichts in den Rückkanal gesendet werden. Die Initialisierung kann auch einer der beteiligten Prozesse übernehmen, zum Beispiel der Erzeuger in Abbildung 2.10, ein Verbraucher in Abbildung 2.11.

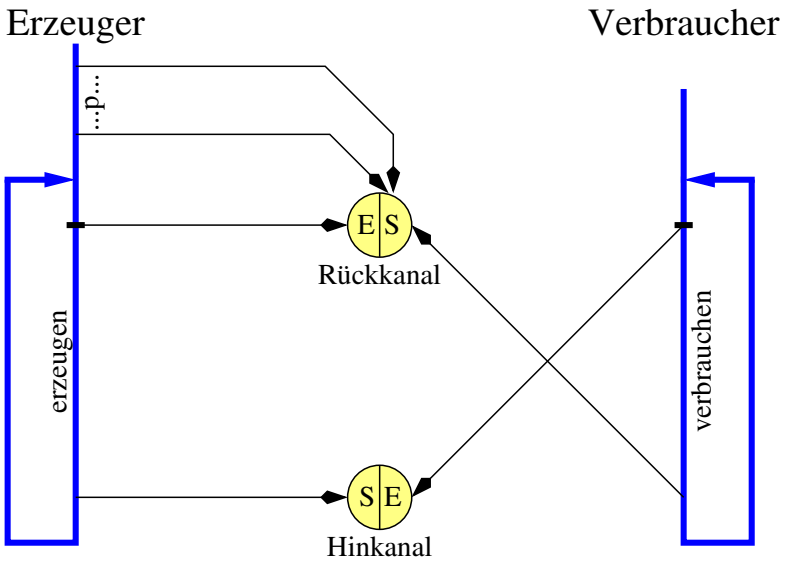


Abbildung 2.10.: Initialisierung durch Erzeuger

Die Darstellung mit Kanälen ist hier gewählt, um das Interaktionsmuster mit den anderen vergleich- und kombinierbar zu halten. Konkrete Implementierungen findet man eher in Form von Pufferverwaltungen, die Schreiber bzw. Leser blockieren, wenn es gerade keine leeren bzw. vollen Puffer gibt. Beim oben genannten Beispiel TCP gibt es keine initialen Nachrichten, weil die Größe des Empfangsfensters beim Verbindungsaufbau übermittelt wird. Dort ist auch nicht die Anzahl der Nachrichten oder Datenpakete entscheidend, sondern die Größe der enthaltenen Daten. Das Prinzip ändert sich dadurch nicht.

Erzeuger/Verbraucher (engl.: *producer/consumer*) ist ein geschlossenes System. Damit ist gemeint, dass die Anzahl der zirkulierenden Nachrichten bzw. die maximale Größe der vorproduzierten Daten vom System selbst festgelegt bzw. begrenzt wird. Im Gegensatz dazu sind Client/Server–Systeme tendenziell offen. Solange es keine harten Grenzen für die Anzahl der Clients und der gleichzeitigen Aufträge pro Client gibt, lässt sich nicht vorhersagen, wieviele Aufträge bei einem Server höchstens anstehen. In konkreten Fällen kann es solche Grenzen aber geben, zum Beispiel bei der wilden Client–Server–Kombination in Abbildung 2.7 auf Seite 15. Sofern die Server auf der rechten Seite dort nur Aufträge von den Prozessen in der Mitte bekommen, liegen bei jedem Server höchstens zwei Aufträge gleichzeitig an.

Bei der Darstellung von Erzeuger/Verbraucher–Systemen ebenso wie bei den Fließbändern in Abschnitt 2.5 könnte der Eindruck entstehen, dass Daten aus dem Nichts kommen oder keine Ergebnisse geliefert werden. Das liegt daran, dass die Interaktionsdiagramme nur einen Teil des jeweiligen Systems zeigen. Ein Erzeuger kann Eingaben zum Beispiel aus Dateien, Datenbanken oder von der Peripherie beschaffen. Ebenso kann ein Verbraucher Ausgaben in Dateien oder Datenbanken ablegen, am Bildschirm anzeigen oder als Töne abspielen. Die dazu notwendige Interaktion mit Treibern oder anderen Prozessen ist hier in den Diagrammen nicht dargestellt, weil sie für die abstrakten Interaktionsmuster keine Rolle spielt.

2.3.1. Mehrere Erzeuger und Verbraucher

Abbildung 2.11 zeigt ein Erzeuger/Verbraucher–System mit drei Erzeugern und zwei Verbrauchern. Das sind wieder Beispiele für das Interaktionsmuster *Reproduktion*, wie zuvor der reproduzierte Server in Abbildung 2.6 auf Seite 14. Die Nachrichten der Erzeuger bedeuten Arbeit für die Verbraucher. Beide Verbraucher–Prozesse teilen sich die Arbeit, die am unteren Kanal ankommt. Umgekehrt bedeuten Nachrichten im oberen Kanal Arbeit, oder zumindest das Potential für Arbeit, bei den drei Erzeuger–Prozessen.

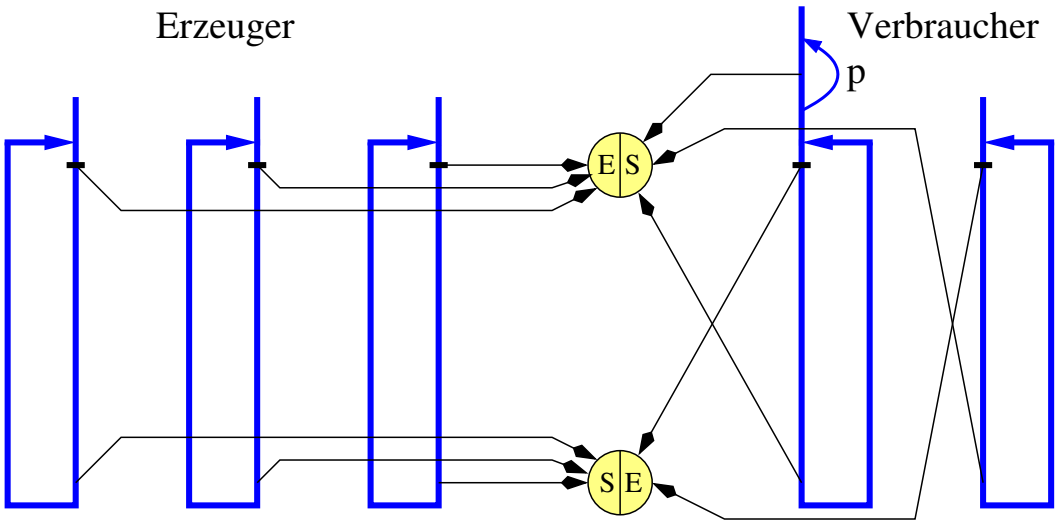


Abbildung 2.11.: Mehrere Erzeuger und Verbraucher

Ein Erzeuger/Verbraucher–System funktioniert dann am besten, wenn beide Seiten ungefähr gleich schnell vorankommen, gleich viele Nachrichten pro Zeiteinheit verarbeiten, den gleichen Durchsatz erreichen. Dazu kann man die Anzahl der Erzeuger und Verbraucher passend einstellen. In der Praxis geschieht das oftmals dynamisch, durch Starten und Beenden von Prozessen passend zur aktuellen oder erwarteten Lastsituation.

2.4. Reproduktion

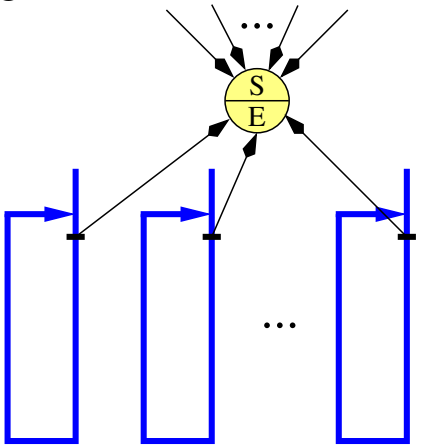


Abbildung 2.12.: Prinzip der Reproduktion

Nachdem die Reproduktion bereits zweimal bei anderen Interaktionsmustern zur Sprache kam, erhält sie hier nun auch einen eigenen Abschnitt. Abbildung 2.12 zeigt das Prinzip. An einem Kanal treffen von irgendwoher Nachrichten ein, die bearbeitet werden müssen. Diese Arbeit teilen sich mehrere, reproduzierte Prozesse. Reproduziert bedeutet, dass alle diese Prozesse die gleiche Software ausführen, also auch die gleiche Logik implementieren. Es spielt keine Rolle, von welchem der Prozesse eine bestimmte Nachricht empfangen und bearbeitet wird. Reproduzierte Prozesse sind austauschbar.
Die Reproduktion ist in mehrfacher Hinsicht ein besonderes Interaktionsmuster. Das zeigt sich zum Beispiel darin, dass Abbildung 2.12 kein vollständiges Interaktionsdiagramm darstellt. Woher die Nachrichten stammen bleibt ebenso unklar wie die Frage, was die Prozesse mit ihnen anstellen. In der Abbildung sieht es so aus, als wären die Prozesse eine Endstation für Nachrichten. Aus den vorangegangenen Beispielen ist aber bekannt, dass reproduzierte Prozesse sehr wohl auch Nachrichten zurück- oder weiterschicken können.
Der Grund für diese Unklarheit ist, dass die Reproduktion nicht für sich alleine eingesetzt werden kann. Zum Reproduzieren braucht man eine Vorlage, also einen Prozess, der eine bestimmte Rolle übernimmt. Zum Beispiel einen Server in Abschnitt 2.2.2, oder Erzeuger und Verbraucher in Abschnitt 2.3.1. Weiter unten folgen noch Fließbandstufen in Abschnitt 2.5.1 und Teammitglieder in Abschnitt 2.7.1. Auch mit anderen, hier nicht vorgestellten Interaktionsmustern spielt die Reproduktion zusammen.
Eine andere Unklarheit in Abbildung 2.12 ist die Anzahl der reproduzierten Prozesse. Tatsächlich besteht ein großer Vorteil der Reproduktion darin, dass man diese Anzahl flexibel konfigurieren oder sogar dynamisch an die aktuelle Lastsituation anpassen kann (engl.: *autoscaling*). Voraussetzung für beides ist, dass die Möglichkeit zur Reproduktion zunächst einmal implementiert wurde.

2.4.1. Parallelität

Einer der Gründe für den Einsatz von Interaktionsmustern ist, dass verschiedene Prozesse parallel arbeiten können. Das ist ein Weg, die verfügbare Hardware besser auszulasten, um anstehende Arbeit schneller zu erledigen. Beim Entwurf paralleler Algorithmen und Anwendungen unterscheidet man zwei Arten der Parallelität, die sich auch kombinieren lassen:

Aufgabenparallelität: engl.: *task parallelism*
Prozesse übernehmen verschiedene Aufgaben oder Arten von Berechnungen.

Datenparallelität: engl.: *data parallelism, loop–level parallelism*
Prozesse führen die gleichen Berechnungen auf verschiedenen Daten aus.

Client/Server und Erzeuger/Verbraucher sind Beispiele für Aufgabenparallelität. Prozesse übernehmen unterschiedliche Rollen, das Potential zur Parallelisierung ist begrenzt. Erzeuger und Verbraucher arbeiten potentiell parallel. Client und Server nur dann, wenn der Client einen asynchronen Auftrag schickt. Will man neue Rollen einführen, um das Potential zur Parallelität zu erhöhen, braucht man neue Software für diese Rollen.
Reproduktion ist ein Beispiel für Datenparallelität. Mehrere Prozesse in der gleichen Rolle können parallel arbeiten, sie führen die gleiche Software aus. Damit schafft man eine Stellschraube, über die man den Grad der Parallelisierung sowohl erhöhen als auch begrenzen kann. Ein anderes Beispiel für Datenparallelität sind Clients. Mehrere Clients können die gleiche Software ausführen und parallel arbeiten. Als Stellschraube eignet sich die Anzahl der Clients eher nicht, wenn sie mit dem Interesse am angebotenen Dienst korreliert. Künstlich aufgebauschte Nachfrage ist ebenso problematisch wie verprellte Benutzer. Viele Clients schaffen aber das Potential, auf der Server–Seite Reproduktion einzusetzen.
Auf einem einzelnen Rechner ist die parallele Ausführung von Prozessen durch die Zahl der Prozessoren beschränkt. Setzt man mehr mehrere Rechner ein, entsteht ein *verteiltes System*. Alle hier vorgestellten Interaktionsmuster eignen sich auch, manche sogar insbesondere, für verteilte Systeme. In einem verteilten System kann es zum Beispiel sinnvoll sein, Teile einer Anwendung in die Clients zu verlagern, um die Server zu entlasten. Steigt die Zahl der Clients durch neue Benutzer, dann erhöht das auch die nutzbaren Hardware–Resourcen auf der Clientseite. In meiner Betriebssysteme–Vorlesung und diesem Buch geht es aber in erster Linie um einzelne Rechner.

2.4.2. Partitionierung

Bei der Reproduktion sind alle reproduzierten Prozesse austauschbar. Es spielt keine Rolle, bei welchem Prozess eine Nachricht bzw. ein Arbeitspaket ankommt. Als Grund hatte ich genannt, dass alle Prozesse die gleiche Software ausführen. Das gilt nicht mehr, wenn die Prozesse nicht nur Algorithmen ausführen, sondern auch auf gespeicherte Daten zurückgreifen. Um größere Datenbestände handhabbar zu halten, können diese in kleinere Partitionen aufgeteilt werden. Für jede Partition sind andere Rechner zuständig, es handelt sich also um ein verteiltes System. Neben der Größe eines Datenbestands können auch andere Gründe die Partitionierung motivieren. Zum Beispiel Zugriffszeiten oder rechtliche Anforderungen: Daten aus Europa auf Rechnern in Europa.

In jeder Partition führen Prozesse die gleiche Software aus. Trotzdem sind sie nicht austauschbar, da sie nur auf einen Teil des gesamten Datenbestands zugreifen können. In einfachen Fällen müssen Arbeitspakete nur zur richtigen Partition geleitet werden. In komplizierteren Fällen muss ein Arbeitspaket auf mehrere Partitionen aufgeteilt und die Teilergebnisse anschließend zusammengeführt werden. So entstehen neue Interaktionsmuster, die nicht zur vorliegenden Sammlung gehören.

2.5. Fließband

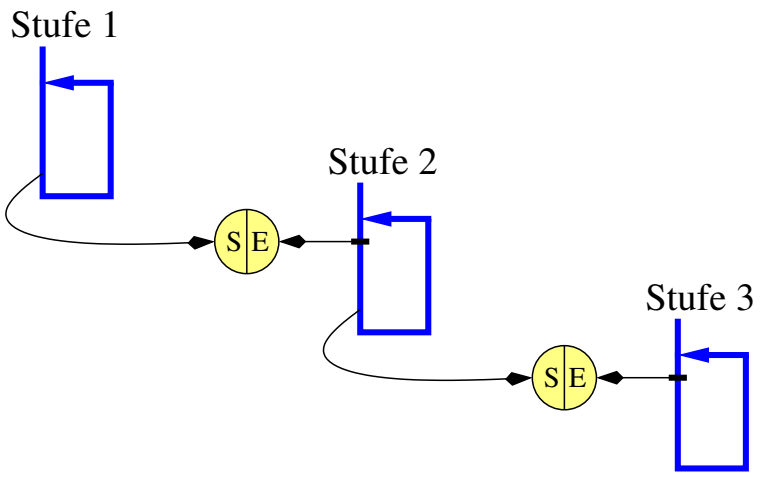


Abbildung 2.13.: Dreistufiges Fließband

Beispiele: C–Quelltext übersetzen: Präprozessor, Compiler, Assembler, Linker. Verkettung von Befehlen mit dem Pipe–Symbol | in Unix–Shells. Datenbankzugriffe: Query parsen, optimieren, Zugriffsplan erstellen.

In einem Fließband (engl.: *pipeline*) sind verschiedene Prozesse für je einen Arbeitsschritt in einer Kette zuständig. Zwischenergebnisse fließen von einer Stufe zur nächsten. Abbildung 2.13 zeigt ein dreistufiges Fließband, mit zwei Kanälen für die Zwischenergebnisse nach der ersten und der zweiten Stufe. Ein zweistufiges Fließband, das nur aus einer Quelle und einer Senke besteht, ist in Abbildung 2.9 auf Seite 16 dargestellt.
Hier wie dort entsteht ein Speicherproblem, falls die erste Stufe Daten schneller liefert, als sie von den folgenden Stufen verarbeitet werden können. Der Durchsatz des Fließbands wird von der langsamsten Stufe begrenzt. Ein dauerhaft betriebenes Fließband funktioniert dann am besten, wenn alle Stufen ungefähr gleich schnell arbeiten. Baut man ein Fließband nur vorübergehend für eine kurze Aufgabe, zum Beispiel in der Kommandozeile einer Shell, spielt die relative Geschwindigkeit der Stufen kaum eine Rolle.
Das Fließband lässt sich mit den bereits vorgestellten Interaktionsmustern kombinieren. Als *offenes Fließband* übernimmt es die Rolle eines Servers, wie in Abbildung 2.14 mit zwei Stufen dargestellt. Die erste Stufe empfängt Aufträge und beginnt mit der Bearbeitung, die letzte Stufe schickt Antworten an die Clients zurück. Dazwischen können auch noch weitere Stufen liegen.

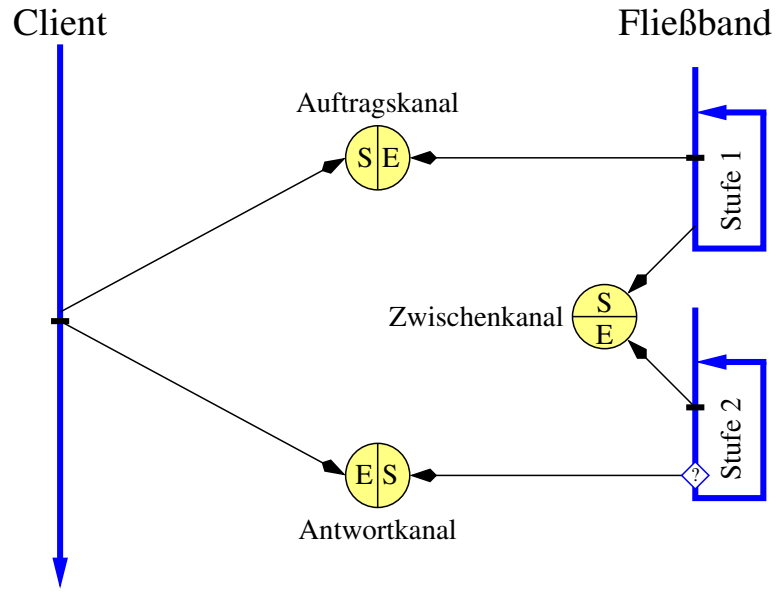


Abbildung 2.14.: Offenes Fließband als Server, zwei Stufen

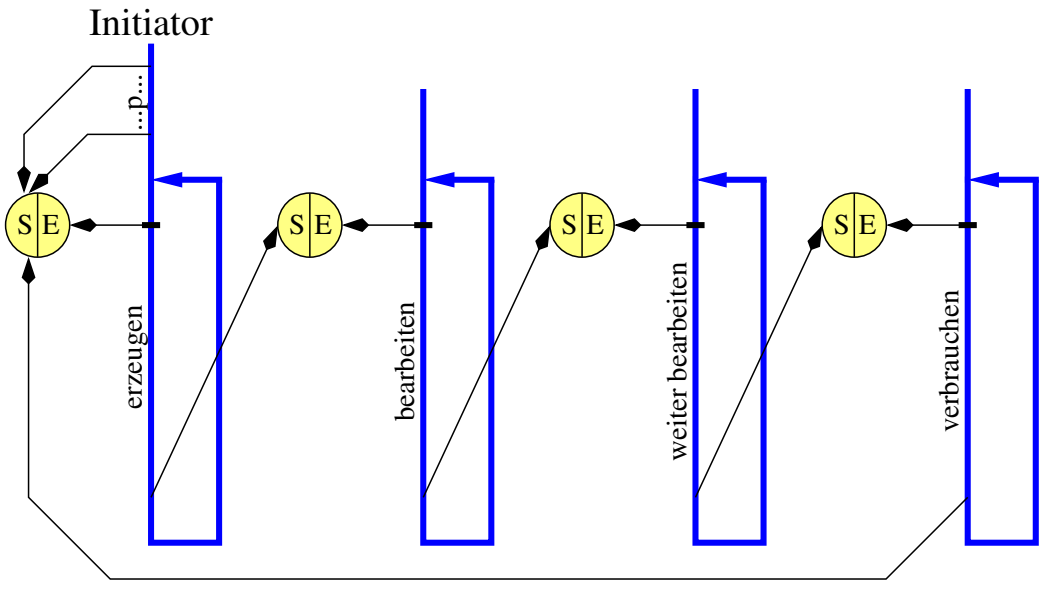


Abbildung 2.15.: Geschlossenes Fließband, vier Stufen

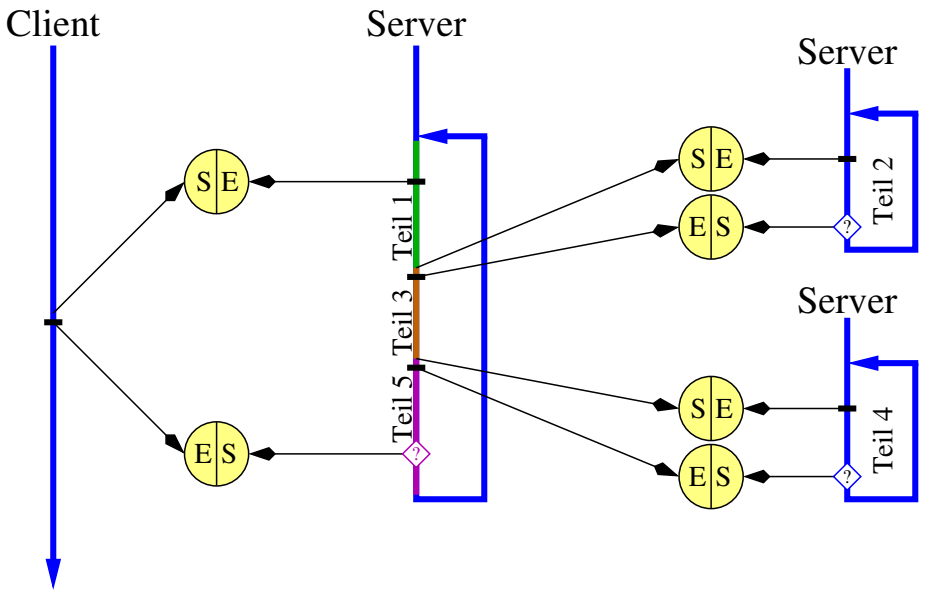


Abbildung 2.16.: Server mit Unteraufträgen

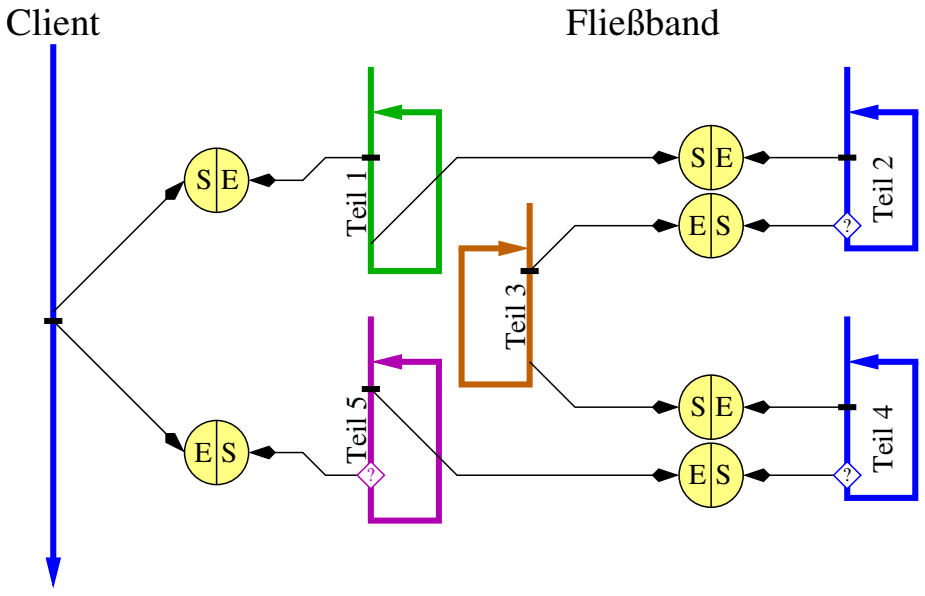


Abbildung 2.17.: Fließband mit eingebundenen Servern

In einem *geschlossenen Fließband* ist die letzte Stufe mit der ersten gekoppelt, wie bei einem Erzeuger/Verbraucher–System. Das einfache Erzeuger/Verbraucher–System aus Abbildung 2.8 auf Seite 16 ist bereits ein zweistufiges, geschlossenes Fließband. Abbildung 2.15 zeigt ein Beispiel mit vier Stufen. Die Anzahl p der Nachrichten oder Puffer sollte hier mindestens so groß sein wie die Anzahl der Stufen, damit alle Prozesse parallel arbeiten können. Auch hier hilft eine größere Anzahl dabei, gelegentliche Schwankungen in der Geschwindigkeit der Stufen auszugleichen.
Der Vorteil eines Fließbands gegenüber einem einzelnen Prozess ist, dass alle Stufen parallel arbeiten und somit mehr Prozessoren nutzen können. Teilt man die Verarbeitungsschritte aber ungünstig auf, klappt das nicht. Kommunikationsverluste durch große Zwischenergebnisse können den Nutzen auffressen, eine einzige lahme Stufe das ganze Fließband ausbremsen. Langsame Stufen kann man allerdings durch Reproduktion ausgleichen.

Will man einen einzelnen Prozess in ein Fließband zerlegen, sollte man nach geeigneten Schnittpunkten in der Verarbeitung suchen. Besonders praktisch ist es, wenn der Prozess Aufträge an Server vergibt und auf Antworten warten muss. Abbildung 2.16 zeigt ein Beispiel. Es bietet sich an, die Aufträge am Ende einer Stufe zu senden und die Antworten in einer anderen Stufe zu bearbeiten. Falls der verwendete Kommunikationsmechanismus das erlaubt, werden die beauftragten Server dann zu weiteren Stufen im Fließband, wie in Abbildung 2.17 dargestellt. Durch Dreiteilung des ersten Servers ist hier ein fünfstufiges Fließband entstanden.
Je länger ein Fließband ist, desto mehr Arbeit kann es parallel erledigen. Allerdings verlängert sich dadurch auch die Einschwingphase. Wenn erste Aufträge eintreffen oder ein geschlossenes Fließband anläuft, kann zunächst nur die erste Stufe arbeiten. Die späteren Stufen müssen warten, bis Zwischenergebnisse dorthin gelangen.
Es ist nicht immer notwendig, dass ein Auftrag oder Puffer sämtliche Stufen eines Fließbands durchläuft. Abhängig von der Art der Bearbeitungsschritte und den konkreten Daten kann es sinnvoll sein, manche Stufen zu überspringen. In einem offenen Fließband könnten dann auch andere Stufen als die letzte Antworten an die Clients schicken.

2.5.1. Reproduktion

Ein Fließband hat für jede Stufe einen Kanal, an dem die Arbeit ankommt. Deshalb kann man in jeder Stufe Reproduktion einsetzen, um mehrere Arbeitspakete parallel zu bearbeiten. Ein Fließband funktioniert dann am besten, wenn alle Stufen etwa gleich schnell arbeiten, den gleichen Durchsatz erreichen. Das kann man durch die Anzahl der reproduzierten Prozesse einstellen. Diese Aussagen erinnern an die zum Erzeuger/Verbraucher–System in Abschnitt 2.3.1. Da es sich dabei um den Spezialfall eines geschlossenen, zweistufigen Fließbands handelt, dürfte das nicht überraschen.
Abbildung 2.18 zeigt ein offenes, dreistufiges Fließband mit reproduzierten Stufen. Die erste Stufe umfasst vier, die zweite drei, die dritte noch zwei Prozesse. Das Diagramm enthält auch eine unbestimmte Anzahl an Clients, die jeweils eine unbestimmte Anzahl an Aufträgen in das Fließband schicken. Die Clients sind nicht *reproduziert*, auch wenn sie die gleiche Software ausführen. Voraussetzung für die Reproduktion als Interaktionsmuster ist, dass sich die reproduzierten Prozesse Arbeit teilen, welche an einem gemeinsamen Kanal ankommt.

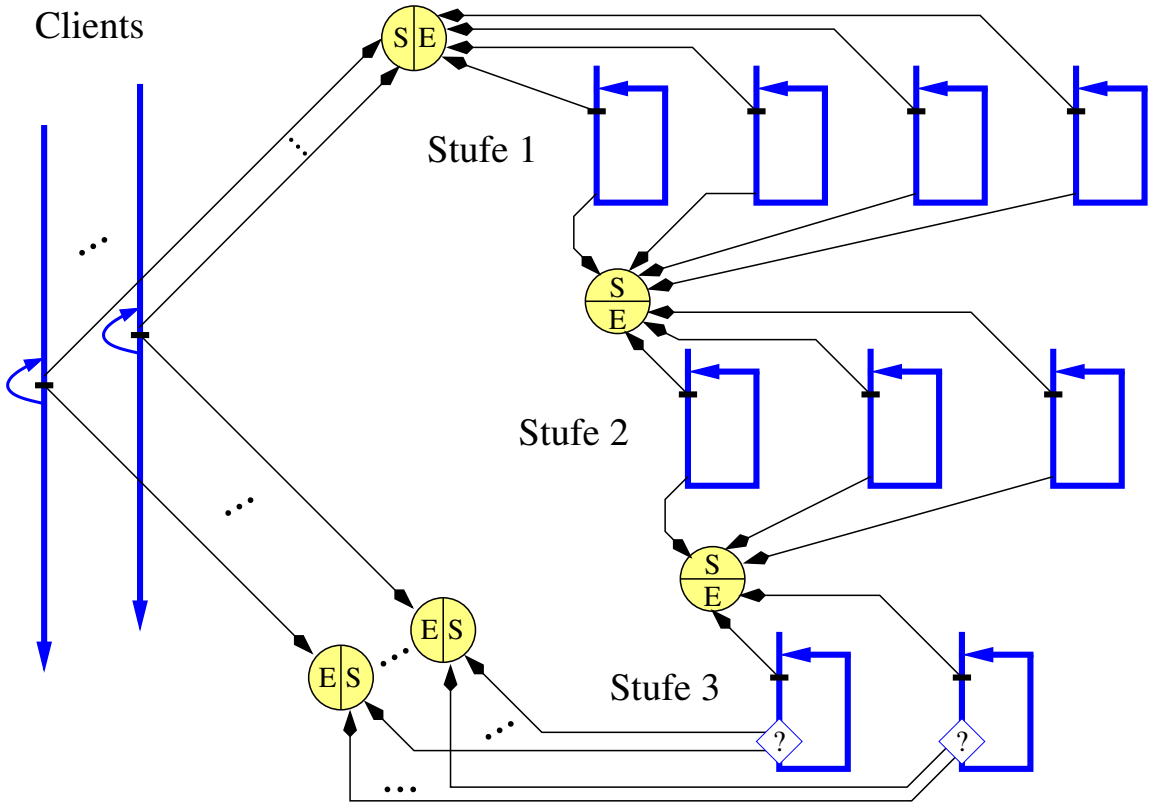


Abbildung 2.18.: Fließband mit reproduzierten Stufen

Hinweis: *Eine Entscheidungsraute ist nur in den Prozessen der letzten Stufe eingezeichnet. Diese müssen entscheiden, an welchen Client sie die Antwort schicken. In den vorhergehenden Stufen gibt es jeweils nur einen Kanal, an den die Prozesse senden.*

2.6. Pufferung

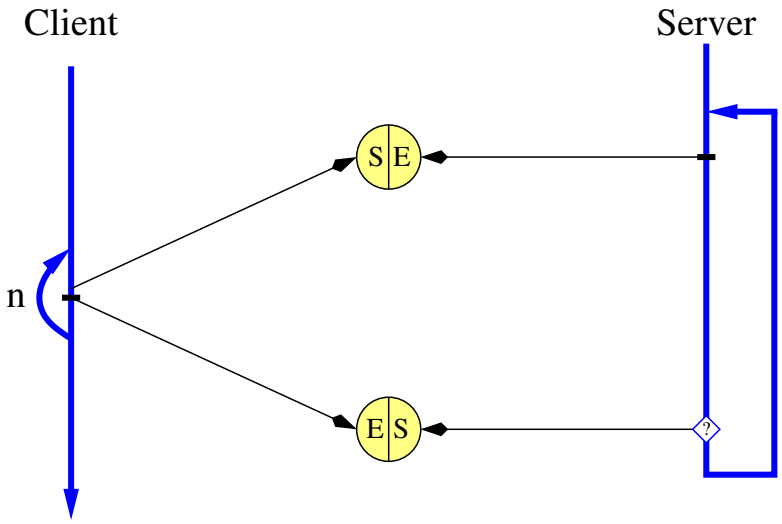


Abbildung 2.19.: Beauftragung in Schleife, ohne Pufferung

Abbildung 2.19 zeigt die Ausgangssituation: ein Client verschickt n Aufträge in einer Schleife. Voraussetzung für die Pufferung ist, dass die Schleifendurchläufe voneinander unabhängig sind. Das heißt, man könnte prinzipiell erst alle Aufträge losschicken und anschließend auf die Ergebnisse warten, wie in Abbildung 2.20. Bei einer großen Zahl von Aufträgen überlastet man damit aber den Server und die Kanalpuffer. Das ist keine gute Idee.

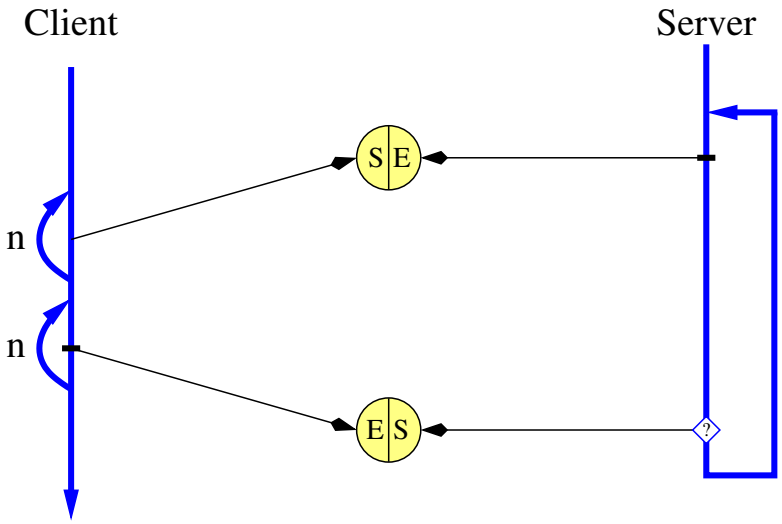


Abbildung 2.20.: Unbeschränkte Pufferung

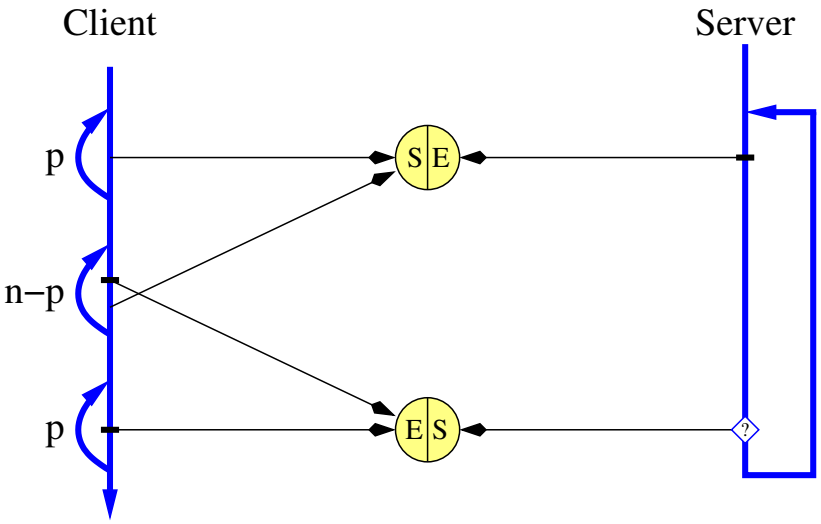


Abbildung 2.21.: Beschränkte Pufferung

Bei der normalen, beschränkten *Pufferung* begrenzt man deshalb die Zahl der ausstehenden Aufträge. Sei n die Zahl der Schleifendurchläufe bzw. Aufträge insgesamt und p das Limit für gepufferte Aufträge, wobei $n \gg p > 1$ gilt. Der Client wird so umstrukturiert, dass er drei Phasen durchläuft. Am Anfang schickt er p Aufträge los. Dann folgt ein Mittelteil, in dem er jeweils auf eine Antwort wartet und dann den nächsten Auftrag vergibt. Wenn der letzte Auftrag gesendet ist, empfängt er in der Schlussphase nur noch die p ausstehenden Antworten. Das Interaktionsdiagramm ist in Abbildung 2.21 zu sehen.
Selbst wenn der Server nur einen Prozess zur Bearbeitung von Aufträgen einsetzt, vermeidet die Pufferung Leerlauf. Richtig nützlich wird sie aber, wenn der Server mehrere Aufträge gleichzeitig bearbeiten kann. Dann erhöht die Pufferung den Parallelitätsgrad auf Serverseite bis zu p.

Praxisbeispiel: (*selbst so implementiert*)
Eine Webanwendung greift auf einen Server zurück, um Nachrichten für die Anzeige im Browser zu formatieren. Nachrichten und das dafür generierte HTML können ziemlich kompliziert werden. Zum Ausdrucken an einem lokalen Drucker soll die Anwendung bis zu 100 Nachrichten auf einer Webseite kombinieren. Das Formatieren und Drucken der Seite übernimmt dann der Browser. Schickt die Anwendung nur einen Request für alle Nachrichten, damit der Server die HTML–Seite kombiniert, braucht der Server Extralogik und viel Speicher. Schickt die Anwendung einen Request pro Nachricht sequentiell, dauert es viel zu lange, bis die Webseite zusammengesetzt ist. Schickt die Anwendung Requests für alle Nachrichten parallel, geht der Server in die Knie. Deshalb beschränkte Pufferung mit fünf parallelen Requests.

2.7. Team

Beispiel Team: Webanwendung mit mehreren Microservices.
Beispiel Sekretär: Webanwendung mit einem Monolithen auf Serverseite.

Ein Team ist eine Gruppe von Prozessen, die jeweils eine andere Aufgabe haben. Zusammen decken diese Spezialisten aber einen gemeinsamen Aufgabenbereich ab. Als Kontrast und zum Vergleich zeigt Abbildung 2.22 kein Team, sondern einen einzelnen Serverprozess, der je nach Bedarf unterschiedliche Aufgaben übernimmt. Tatsächlich ist es ganz normal, dass ein Server eine ganze Auswahl an Funktionen anbietet. Je nach Art des Auftrags führt er mal diese, mal jene Funktion aus. Wettstein nennt das einen *Sekretär*.[1] Die große Raute mit Fragezeichen in Abbildung 2.22 bedeutet, dass der Befehlstrom hier jeweils einen der drei möglichen Wege nimmt, abhängig von der Art des empfangenen Auftrags. Vor dem Senden der Antwort laufen diese Wege wieder zusammen. Die kleine Raute mit Fragezeichen steht dann für die Auswahl des Antwortkanals.
Eine Möglichkeit zur Parallelisierung des Sekretärs ist, für verschiedene Auftragstypen jeweils einen eigenen Prozess mit eigenem Auftragskanal einzurichten. Diese Prozesse bilden dann ein Team. Allerdings muss nun vorher entschieden werden, in welchen Kanal ein Auftrag gehört. Dazu gibt es zwei Ansätze. Will man einen gemeinsamen Auftragskanal nach außen behalten, ergänzt man das Team um einen vorgeschalteten *Verteiler*, der die eingehenden Aufträge zum jeweils passenden Kanal weiterleitet. Das ist in Abbildung 2.23 dargestellt. Im Beispiel des Webbrowsers wäre der Verteiler ein Reverse Proxy, der verschiedene Microservices über einen gemeinsamen Hostnamen anbietet.
Der andere Ansatz ist, die Kanäle der Teammitglieder von außen zugänglich zu machen und die Entscheidung an die Stelle zu verlagern, wo der Auftrag geschickt wird. Dazu könnte man zum Beispiel eine Bibliothek bereitstellen. Oder ein Webbrowser führt Code aus, den er zuvor vom Webserver geladen hat. Abbildung 2.24 zeigt diese Variante, die Wahl des Auftragskanals findet im Client statt.

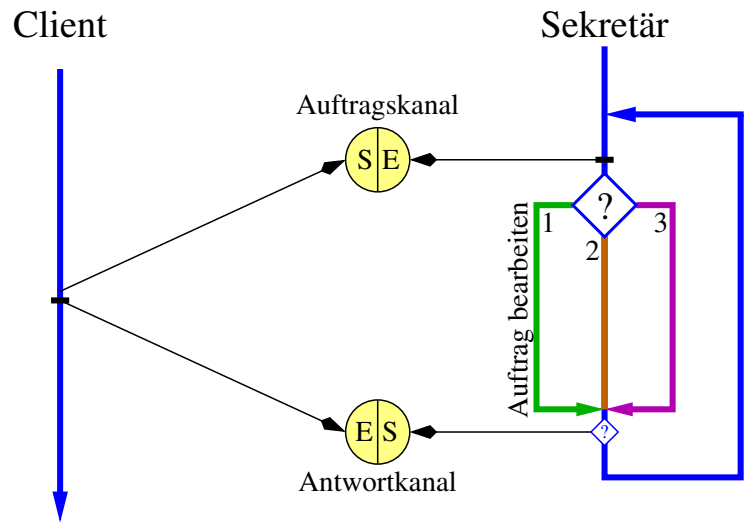


Abbildung 2.22.: Ein Sekretär erledigt verschiedene Aufgaben

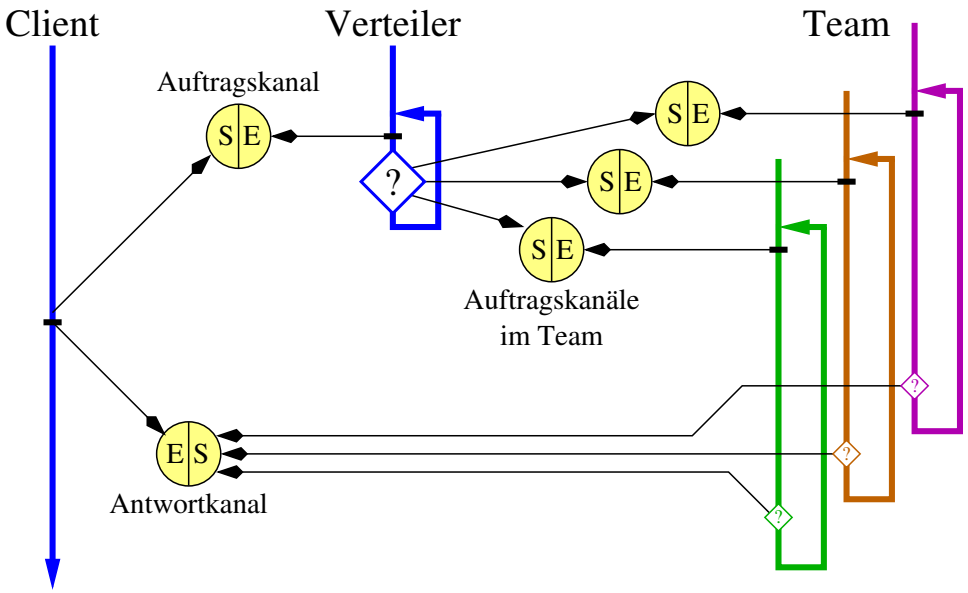


Abbildung 2.23.: Team mit Verteiler

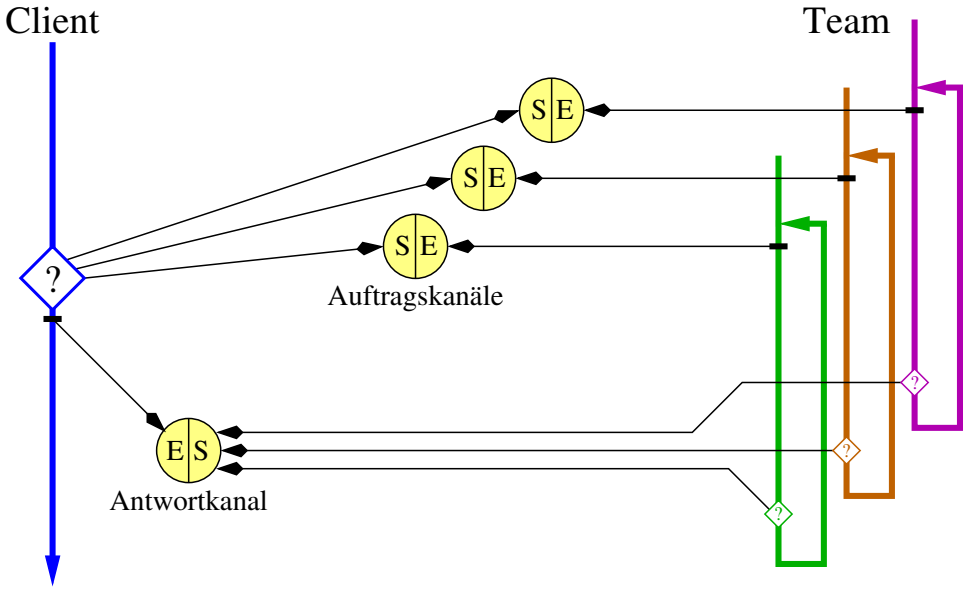


Abbildung 2.24.: Team ohne Verteiler

Für verbindungsorientierte Kommunikation passt das Interaktionsdiagramm des Teams mit Verteiler in Abbildung 2.23 nicht mehr richtig. Der Client baut eine Verbindung zum Verteiler auf und erwartet auch die Antwort über diese Verbindung. In manchen Situationen ist es denkbar, offene Verbindungen von einem Prozess zu einem anderen auf dem gleichen Rechner weiterzureichen. Dann könnte der zuständige Teamprozess die Antwort direkt an den Client schicken. Häufiger ist allerdings der Fall, dass der Verteiler seinerseits eine Verbindung zum Teamprozess aufbaut. Dann fließt auch die Antwort vom Teamprozess über den Verteiler zurück zum Client. So arbeiten Webproxies.
Auch in einem Team kann jeder Prozess ein Sekretär sein, der verschiedene Untertypen von Aufträgen erledigt. Für die Interaktionsdiagramme ist das aber nicht weiter interessant, da sie vor allem den Fluss der Nachrichten und die Beziehungen zwischen Prozessen darstellen sollen. In der Praxis muss man ein gesundes Mittelmaß für die Aufgabenteilung im Team finden, ebenso wie bei anderen Interaktionsmustern.

2.7.1. Reproduktion

In einem Team gibt es für jede Art von Auftrag einen eigenen Kanal. Dementsprechend kann man die Reproduktion pro Auftragstyp einsetzen. Abbildung 2.25 zeigt ein Team mit Verteiler für drei Arten von Aufträgen: A, B und C. Um jede Art von Auftrag kümmern sich zwei reproduzierte Prozesse. Im Gegensatz zu den Stufen eines Fließbands geht es hier nicht darum, dass alle Arten von Aufträgen etwa gleich schnell bearbeitet werden. Vielmehr sollte die Einstellung danach erfolgen, mit welcher Häufigkeit welche Art von Auftrag eintrifft.
Prinzipiell kann man auch den Verteiler reproduzieren. In der Praxis kann das sinnvoll sein, zum Beispiel wenn die Gefahr besteht, dass ein überfüllter Kanal den sendenden Prozess blockiert. In den Interaktionsdiagrammen gehe ich aber von der Annahme aus, dass ein Verteiler nur Nachrichten weiterreicht und nicht zum Engpass wird. Dann hat es wenig Nutzen, das Diagramm durch einen reproduzierten Verteiler komplizierter zu machen.

Gelegentlich kommt es vor, dass jemand die Interaktionsmuster *Team* und *Reproduktion* verwechselt. Die Unterschiede treten in Abbildung 2.25 besonders deutlich zu Tage. Ein Team besteht aus Spezialisten für *unterschiedliche* Arten von Aufträgen. Jedes Mitglied führt andere Software aus, für jede Art von Auftrag gibt es einen eigenen Kanal. Reproduzierte Prozesse sind austauschbar. Sie bearbeiten die *gleiche* Art von Aufträgen, sie führen die gleiche Software aus, sie empfangen die Aufträge an einem gemeinsamen Kanal. Durch die Kombination von Team und Reproduktion gibt es mehrere Prozesse, welche die Rolle eines bestimmten Spezialisten übernehmen können.

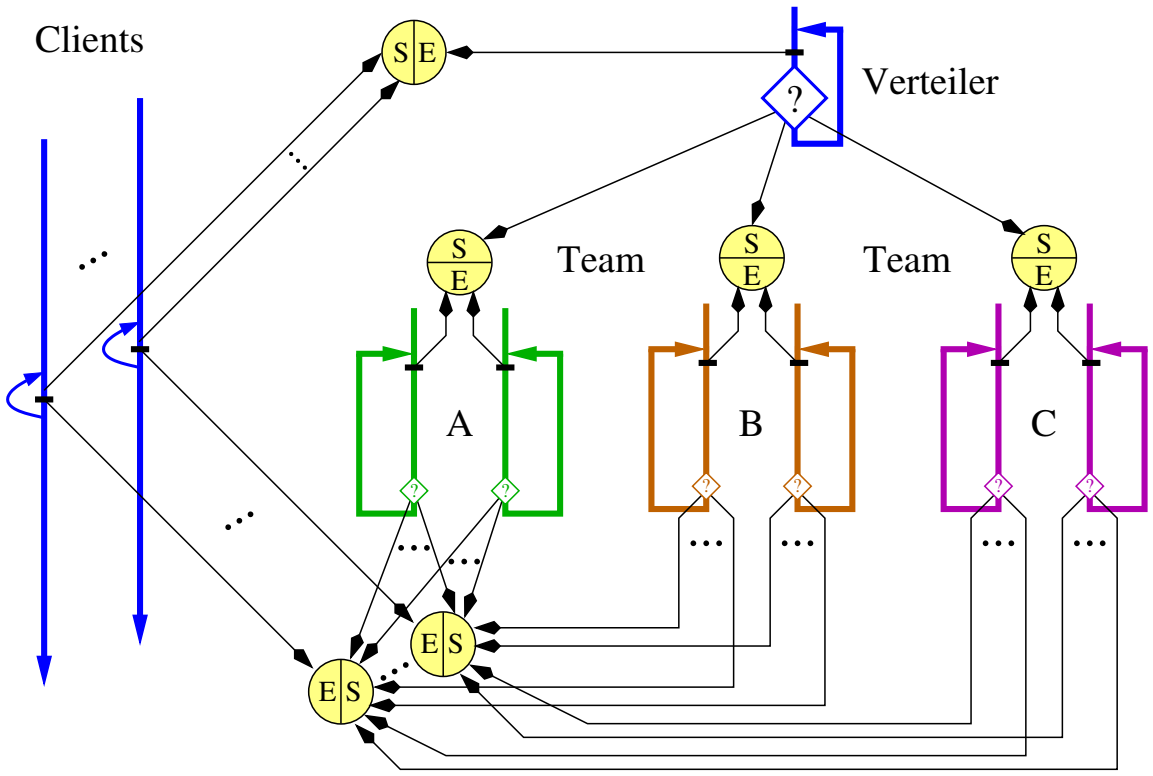


Abbildung 2.25.: Team mit Reproduktion

2.8. Knackpunkte

- Instanzen bzw. Prozesse innerhalb von Instanzen sind Module. Interaktion verbindet diese Art von Modulen.
- Interaktionsmuster sind Vorlagen, um die Zusammenarbeit von Prozessen zu strukturieren. Die hier vorgestellten Muster sind nur eine kleine Auswahl.
- Interaktionsdiagramme sind ein Weg, manche Interaktionsmuster darzustellen, aber längst nicht alle. Siehe Abschnitt 2.8.1 für ein Beispiel, bei dem sie nicht helfen.
- Client/Server verbindet Prozesse hierarchisch. Clients schicken Aufträge und erwarten jeweils eine Antwort. Server empfangen Aufträge, bearbeiten sie, und schicken Antworten.
  - Asynchrone Aufträge sind solche, bei denen der Client zunächst noch weiterarbeitet und erst später auf die Antwort wartet.
  - C/S–Systeme sind offene Systeme. Kontrolle gibt es meist nur über den Server, aber nicht über die Clients. Zu viele gleichzeitige Aufträge von Clients können den Server überlasten.
- Erzeuger/Verbraucher verbindet Prozesse auf Augenhöhe. Erzeuger fangen etwas an, Verbraucher bringen es zu Ende.

  - E/V–Systeme sind geschlossene Systeme. Es zirkulieren eine begrenzte Anzahl von Nachrichten oder Puffern. Im besten Fall ist damit auch der Resourcenbedarf des Systems begrenzt.
  - E/V–Systeme müssen in Schwung gebracht werden.

- Reproduktion ermöglicht Datenparallelität in anderen Interaktionsmustern.
  - Mehrere Prozesse in der gleichen Rolle (Server, Erzeuger, Verbraucher,. . .) teilen sich Arbeit, die an einer bestimmten Stelle anfällt.
  - Mehrere Clients sind keine Reproduktion. Clients teilen sich keine Arbeit, die an einer bestimmten Stelle anfällt.
  - Reproduzierte Prozesse führen die gleiche Software aus. Mehrfache Reproduktion der gleichen Rolle erfordert keinen zusätzlichen Entwicklungsaufwand. Es genügen Konfigurationseinstellungen.
  - Autoskalierung bedeutet, dass der Grad der Reproduktion automatisch angepasst wird, zum Beispiel an die Lastsituation.

- Ein Fließband kombiniert zwei oder mehr Prozesse in Serie. Ein offenes Fließband spielt die Rolle eines Servers. Geschlossene Fließbänder arbeiten wie E/V–Systeme, ggfs. mit mehr als zwei Stufen.

- Mittels Pufferung schickt ein Client mehrere Aufträge überlappend. Das ermöglicht Datenparallelität auf der Server–Seite, für diesen einzelnen Client.

- Ein Team besteht aus verschiedenen Rollen, die auf unterschiedliche Arten von Aufträgen spezialisiert sind. Aufträge müssen bei der richtigen Rolle ankommen. Das kann ein Verteiler übernehmen, oder Logik beim Sender der Aufträge.

2.8.1. Grenzen der Darstellung

Die in diesem Kapitel verwendeten Interaktionsdiagramme funktionieren nur, wenn man die Interaktion zwischen verschiedenen Rollen direkt auf die Kommunikation an verschiedenen Kanälen herunterbrechen kann. Ein Interaktionsmuster, bei dem das nicht funktioniert, ist *Publish/Subscribe*. Hier gibt es drei Rollen:

- Ein Message Broker vermittelt zwischen Publishern und Subscribern.

- Publisher schicken Meldungen zu bestimmten Themen (engl.: *topics*) an den Broker.

- Subscriber melden beim Broker Interesse an bestimmten Themen an und erhalten dann nur Meldungen zu diesen Themen.

- Im Broker kümmern sich verschiedene Prozesse um die Verwaltung der Interessen und um das Weiterleiten der Meldungen.

Die Beziehungen zwischen Publishern, Broker und Subscribern sind statisch und könnten wohl in ein Interaktionsdiagramm gegossen werden. Das würde durch die verschiedenen Prozesse im Broker aber unübersichtlich. Die eigentlich relevanten Interaktionsbeziehungen spielen sich jedoch zwischen den Publishern und Subscribern einzelner Themen ab. Hier sind die Möglichkeiten der Darstellung am Ende. Man müsste ein neues Interaktionsobjekt „Thema“ einführen, auf einer viel höheren Abstraktionsebene als die Kanäle.

3. Software

Die vielen verschiedenen Programmiersprachen, in denen Software entwickelt wird, kann kein Prozessor direkt verarbeiten. Jede Prozessorarchitektur (ARM, PowerPC, x86) definiert einen Befehlssatz, mit einem festgelegten Binärformat. Prozessoren lesen Sequenzen dieser Befehle aus dem Hauptspeicher und führen sie aus. So entstehen Befehlsströme, die die gewünschte Datenverarbeitung durchführen. Software in Form von prozessorspezifischen Befehlen bezeichnet man als „nativen“ Code oder Binärcode.
Im Wesentlichen führen zwei Wege von Programmiersprachen zu Binärcode: Übersetzen oder Interpretieren. Ein Compiler analysiert die Software und übersetzt sie vorab in ein anderes Format, oft in Binärcode, das später zur Ausführung gebracht wird. Ein Interpreter analysiert die Software zur Laufzeit und ruft eigene Routinen in Binärcode auf, um das gewünschte Verhalten zu erzielen. In beiden Fällen liegt zur Laufzeit Binärcode vor, entweder der der Software oder der des Interpreters.
Typische übersetzte Sprachen sind C, C++ und Assembler. Typische interpretierte Sprachen sind Python, JavaScript und Lisp. Zudem gibt es Mischformen und Sonderfälle. Ein Java–Compiler erzeugt aus dem Quelltext Java–Bytecode, welcher zur Laufzeit von einer JVM interpretiert wird. Die JVM–Implementierung kann einen JIT–Compiler (engl.: *Just In Time*) mitbringen, der zur Laufzeit Teile des Bytecodes in nativen Code übersetzt. Kotlin–Compiler existieren für verschiedene Ziele: Java–Bytecode, JavaScript und Binärcode für mehrere Architekturen. Compiler, die in eine andere Programmiersprache übersetzen, heißen auch Transpiler. Ein weiteres Beispiel dafür ist TypeScript, welches nach JavaScript übersetzt wird.
Unter einem *Programm* verstehe ich im Folgenden eine bereits übersetzte Software, die als ausführbare Datei im passenden Binärformat vorliegt. Bei interpretierter Software ist der Interpreter das Programm. *Bibliotheken* sind ein wichtiges Mittel bei der Entwicklung modularer Software. Sie stellen Implementierungen von Programmierschnittstellen bereit, worauf Kapitel 6 näher eingeht. Hier stehen erst einmal die technischen Aspekte beim Einsatz von Bibliotheken im Vordergrund.
Abschnitt 3.1 beschreibt ein Programm zur Laufzeit im Speicher, also eine Instanz. Abschnitt 3.2 erklärt das Erstellen von ausführbaren Dateien durch Compiler und Linker. Dabei spielen auch statische Bibliotheken eine Rolle. Abschnitt 3.3 geht auf dynamische Bibliotheken ein. Abschnitt 3.4 beleuchtet die Rolle des Programmladers, der aus ausführbaren Dateien und dynamischen Bibliotheken Instanzen erzeugt. Abschnitt 3.5 bespricht Plugins, welche zur Laufzeit nachgeladen werden. Abschnitt 3.6 fasst die wichtigsten Punkte dieses Kapitels zusammen.

3.1. Segmente

Jedes laufende Programm verwendet Teile des Hauptspeichers, auch RAM (engl.: *Random Access Memory*) genannt. Die Speicherinhalte gruppiert man nach Einsatzzweck und Verwaltungsstrategie in mehrere *Segmente*. Jedes Segment umfasst einen größeren, meist zusammenhängenden Bereich des Speichers. Zum Verständnis genügen die folgenden fünf Segmente:

Code: Enthält Befehlssequenzen, die ein Prozessor ausführen kann. Hier stehen übersetzte Funktionen, Prozeduren und Methoden.

Konstanten: Enthält Daten, die nicht verändert werden. Zum Beispiel die Zeichenkette `"Hallo, Welt!"`, falls sie im Programm vorkommt.

Statische Daten: Enthält Daten fester Größe, deren Inhalt sich ändern darf. Der Platz ist während des gesamten Programmlaufs reserviert. Hier landen Variablen und Attribute, die als `static` deklariert sind.

Stapel: Enthält lokale Variablen, Aufrufargumente, Rücksprungadressen und mehr, was zur Abwicklung von verschachtelten Prozedur- oder Methodenaufrufen notwendig ist. Jeder Befehlsstrom braucht einen eigenen Stapel.

Halde: Enthält Daten unterschiedlicher Größe, für die erst während des Programmlaufs Platz angefordert wird. Zum Beispiel belegen die Operationen `new` und `malloc` Speicher auf der Halde.

Abbildung 3.1 zeigt schematisch eine Instanz mit diesen Segmenten. Konstanten und statische Daten sind mit K bzw. D abgekürzt, um diese Bereiche nicht überproportional groß zu zeichnen. Die Proportionen in der Abbildung sind aber trotzdem nicht aussagekräftig.

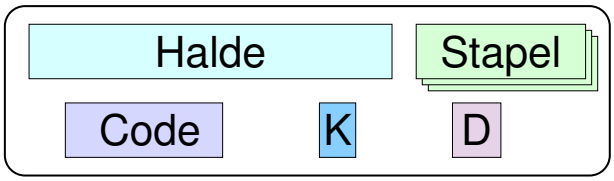


Abbildung 3.1.: Segmente einer Instanz (Auswahl)

Die Größe von Code, Konstanten und statische Daten ist bereits beim Erstellen des ausführbaren Programms bekannt. Zur Laufzeit braucht es keine Verwaltung mehr. Zum Beispiel liegen alle Prozeduren und Methoden hintereinander im Code–Segment. Die vom Compiler generierten Aufrufbefehle springen an die richtige Stelle dort. Ebenso erzeugt der Compiler Befehle, die auf die richtigen Stellen in den Segmenten mit Konstanten bzw. statische Daten zugreifen.
Stapel und Halde dagegen müssen zur Laufzeit verwaltet werden. Instanzen bekommen oder holen sich vom Betriebssystem größere Speicherbereiche für diese Segmente und unterteilen diese passend.

3.1.1. Stapel

Innerhalb eines Befehlsstroms sind Aufrufe von Prozeduren oder Methoden sauber verschachtelt. Jeder Rücksprung beendet den gerade aktuellsten Aufruf. Das nennt man LIFO: *last in, first out*. Auch eine *Exception* (dt.: Ausnahme) folgt diesem Muster, mit dem Unterschied, dass sie mehrere Ebenen von Aufrufen beenden kann, bis sie letztlich gefangen wird.
Die LIFO–Schachtelung macht es möglich, den Speicher für lokale Variablen einer Prozedur oder Methode besonders einfach zu verwalten, nämlich mit einem Stapel (engl.: *stack*). Im Stapel–Segment gibt es genau zwei Bereiche, den belegten und den freien Teil des Stapels. Ein Stapelzeiger definiert die Grenze, wie in Abbildung 3.2 dargestellt. Bei jedem Aufruf holt sich die aufgerufene Prozedur oder Methode ein Stück vom freien Bereich, ihren sogenannten *Stack Frame*. Dazu zählt sie den Stapelzeiger um so viele Byte herunter,[1] wie sie benötigt. Beim Rücksprung setzt sie den Stapelzeiger wieder auf den vorherigen Wert. Die Befehle zum Ändern des Stapelzeigers generiert der Compiler. Dabei wird davon ausgegangen, dass der Stapelbereich groß genug ist.

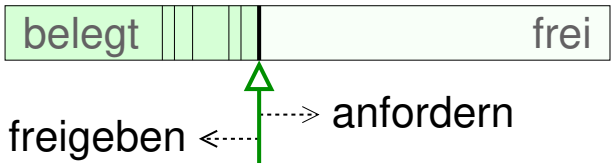


Abbildung 3.2.: Verwaltung eines Stapels

Ist der freie Bereich des Stapels aufgebraucht, liegt ein *Stack Overflow* (dt.: Stapelüberlauf) vor. In guten Fällen erkennt die Infrastruktur dieses Problem und löst einen Fehler aus, der den Befehlsstrom abbricht. Meist ist das die Folge einer Endlosrekursion, also eines Programmierfehlers. Seltener braucht ein Algorithmus tatsächlich einen außergewöhnlich großen Stapel für besonders tief verschachtelte Aufrufe. Dann kann man vor dem Start des Programms die Systemeinstellung für die Stapelgröße anpassen.
Verschiedene Befehlsströme führen ihre Aufrufe und Rücksprünge unabhängig voneinander durch. Die LIFO–Schachtelung gilt nur innerhalb eines Befehlsstroms. Deshalb braucht jeder Befehlsstrom seinen eigenen Stapel. Instanzen mit mehreren Befehlsströmen können dazu mehrere Stapel–Segmente verwenden, oder weitere Stapel zum Beispiel auf der Halde anlegen. Im letzteren Fall kann die Infrastruktur allerdings nicht erkennen, ob der Stapel überläuft.

3.1.2. Halde

Speicher, der zur Laufzeit angefordert wird und nach Ende des aktuellen Aufrufs noch verfügbar sein soll, stammt von der Halde (engl.: *heap*). Hier gibt es keine festgelegte Reihenfolge, in der angeforderter Speicher wieder frei wird. Das macht die Verwaltung deutlich aufwendiger. Dafür sind Bibliotheksfunktionen zuständig, die eng an die verwendete Programmiersprache gekoppelt sind.

[1] traditionell wachsen Aufrufstapel nach unten, zu niedrigeren Adressen

Typische Funktionen zum Anfordern von Speicher auf der Halde sind `malloc` (C) oder `new` (C++, Java). Wenn Speicher explizit wieder freigegeben werden muss, gibt es dazu Gegenstücke wie `free` oder `delete`. Bei deren Verwendung ist Vorsicht geboten. Ein typischer Programmierfehler besteht darin, Speicher freizugeben, auf den später noch zugegriffen wird.
Eine Alternative zum expliziten Freigeben ist die *Garbage Collection* (*GC*, dt.: Speicherbereinigung). Hier versucht die Verwaltungslogik automatisch zu erkennen, welche Teile der Halde noch in Gebrauch sind. Dabei kann Speicher als noch verwendet eingestuft werden, obwohl die Instanz nicht mehr darauf zugreift. Das führt zu sogenannten Speicherlöchern (engl.: *memory leaks*), also unnötig hohem Speicherbedarf.
Sowohl für die Verwaltung mit expliziter Freigabe als auch mit Garbage Collection gibt es viele verschiedene Strategien, die abhängig vom Verhalten der Instanz mehr oder weniger effizient funktionieren. Allen gemeinsam ist, dass sie große Segmente von der Infrastruktur erhalten und diese selbst in kleineren Stücken verwalten. Im Gegensatz zum Stapel ist es hier auch problemlos möglich, bei Bedarf weiteren Speicher für die Halde anzufordern. Dabei können auch mehrere Haldensegmente entstehen, die nicht zusammenhängen.[2]

3.2. Ausführbare Dateien

Eine direkt ausführbare Datei beschreibt den Speicherinhalt eines Programms zum Zeitpunkt des Programmstarts. Dementsprechend finden wir auch in ausführbaren Dateien einige der in Abschnitt 3.1 vorgestellten Segmente: Code, Konstanten, statische Daten. Außerdem legen sie eine *Einsprungadresse* fest. Diese bestimmt, an welcher Stelle im Code der Befehlsstrom des Programms anfängt.
Um Platz zu sparen, unterteilt man die statischen Daten nochmals. Nicht initialisierte und mit Null initialisierte Daten bilden ein eigenes Segment, getrennt von denjenigen statischen Daten, die mit anderen Werten initialisiert sind. Für dieses oft BSS (engl.: *block storage segment*) genannte Segment steht in der ausführbaren Datei nur die Größe. Halde und Stapel stehen nicht in der ausführbaren Datei. Diese Segmente sind beim Start des Programms leer und haben keine feste Größe.

3.2.1. Binden

Direkt ausführbare Dateien werden von einem *Linker* (dt.: Binder) erstellt, der mehrere Objektdateien kombiniert. Die Objektdateien erzeugt ein *Compiler* (dt.: Übersetzer), indem er einzelne Dateien aus Quelltext der jeweiligen Programmiersprache übersetzt. Eine *statische Bibliothek* ist eine Sammlung von thematisch gruppierten Objektdateien in einem Archiv. Abbildung 3.3 zeigt das Erstellen einer ausführbaren Datei am Beispiel von C. Auch C++ arbeitet nach dem gleichen Prinzip. Jüngere Programmiersprachen wie Rust oder Go verwenden bequemere Werkzeuge, welche automatisch beide Schritte ausführen. Unter der Decke funktionieren sie genauso, nur bleiben die Objektdateien versteckt.
Der Quelltext größerer Programme ist in mehrere Dateien aufgeteilt, die einzeln übersetzt werden können, aber aufeinander Bezug nehmen. Jede Datei kann Klassen, Funktionen

[2] *Overview of Malloc* `https://sourceware.org/glibc/wiki/MallocInternals`, abgerufen 2022-07-09

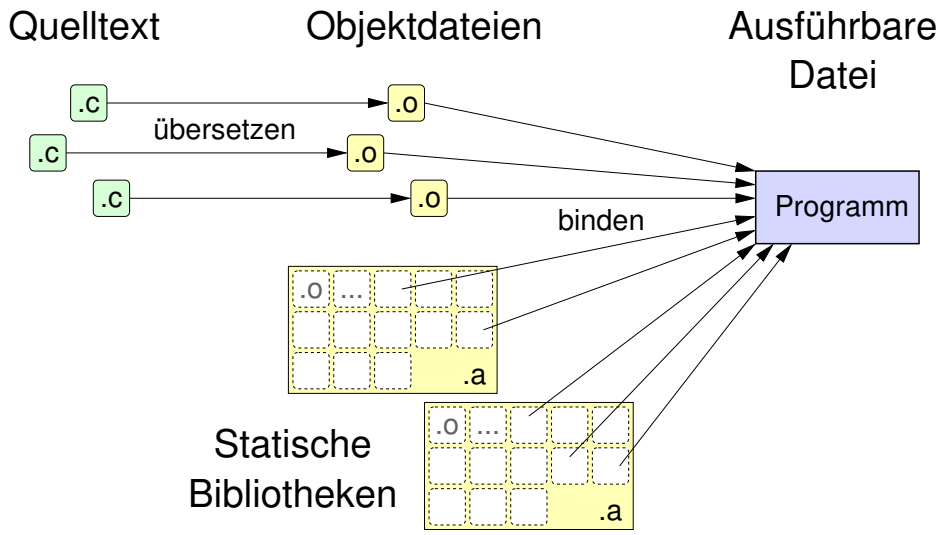


Abbildung 3.3.: Erstellen einer ausführbaren Datei

oder Variablen verwenden, die in einer der anderen Dateien definiert sind. Eine Objektdatei enthält Code, Konstanten und statische Daten, die der Compiler für die jeweilige Quelltextdatei erzeugt. Ihr Format ähnelt dem einer ausführbaren Datei, aber ohne Einsprungadresse. Außerdem definieren und verwenden Objektdateien *Symbole*, die für Klassen, Funktionen oder Variablen des Programms stehen.

Exportierte Symbole sind in der Objektdatei definiert und anderswo verwendbar.

Referenzierte Symbole werden in der Objektdatei verwendet und anderswo definiert.

Der Linker löst sämtliche referenzierten Symbole auf, die in den einzeln übergebenen Objektdateien stehen. Aus den statischen Bibliotheken holt er dazu nur diejenigen Objektdateien, die ein referenziertes Symbol definieren. Diese können wiederum Symbole referenzieren und weitere Objektdateien mit in das Programm ziehen. Findet der Linker ein referenziertes Symbol, das er nicht auflösen kann, meldet er einen Fehler.
Der Linker verschmilzt die Code–Segmente aller Objektdateien zu einem einzelnen Code–Segment in der ausführbaren Datei. Ebenso verfährt er mit den Konstanten, statische Daten und BSS–Segmenten. Als Einsprungadresse dient ein Symbol aus einer speziellen Objektdatei, die der Linker eigenständig hinzufügt. In der ausführbaren Datei ist also das jeweilige Programm mit den benötigten Teilen der statischen Bibliotheken untrennbar kombiniert. Erscheint eine neue Version einer statischen Bibliothek, die zum Beispiel Fehler behebt oder Sicherheitslücken schließt, muss man die ausführbare Datei neu binden, um davon zu profitieren.

3.3. Dynamische Bibliotheken

Es bringt Vorteile, wenn man Programme und Bibliotheken separat installieren und aktualisieren kann. Erst beim Start oder zur Laufzeit wird das Programm mit den aktuellen benötigten Bibliotheken gebunden, in der jeweiligen Instanz. Das nennt man *dynamisches* Binden. Die Bibliotheken müssen dazu in einem anderen Format vorliegen als oben beschrieben. Deshalb unterscheidet man zwischen dynamischen und statischen Bibliotheken. Der englische Begriff *shared libraries* für dynamische Bibliotheken weist auf einen weiteren Vorteil hin. Wenn viele Programme die dynamischen Bibliotheken gemeinsam nutzen, statt jeweils eigene Kopien der verwendeten Bibliotheksfunktionen mitzubringen, spart man Platz auf der Platte und im Speicher. Abbildung 3.4 vergleicht einige Eigenschaften dynamischer und statischer Bibliotheken.

	Dynamische Bibliotheken	**Statische Bibliotheken**
Dateiendung	`.so` `.DLL`	`.a` `.LIB`
Zeitpunkt des Bindens	Start der Instanz oder zur Laufzeit	Erstellen der Programmdatei
Geladener Umfang	komplette Bibliothek separat vom Programm	nur benötigte Funktionen als Teil des Programms
Speicherbedarf	Code und Konstanten nur einmal für alle Instanzen	Code und Konstanten pro geladener Programmdatei

Abbildung 3.4.: Vergleich dynamischer und statischer Bibliotheken

Beim Binden einer ausführbaren Datei löst der Linker alle referenzierten Symbole der übergebenen Objektdateien auf. Findet er ein Symbol in einer statischen Bibliothek, dann nimmt er die Objektdatei aus der Bibliothek mit in die ausführbare Datei auf, wie in Abschnitt 3.2.1 beschrieben. Findet er ein Symbol in einer dynamischen Bibliothek, dann schreibt der Linker sowohl das referenzierte Symbol als auch Name und Version der dynamischen Bibliothek in die ausführbare Datei. Für die Ausführung muss später die passende dynamische Bibliothek geladen sein. Findet der Linker ein referenziertes Symbol weder in den Objektdateien, noch in einer statischen oder dynamischen Bibliothek, löst er einen Fehler aus.[3]
Eine dynamische Bibliothek wird ebenfalls vom Linker erstellt. Ähnlich wie eine ausführbare Datei enthält sie Segmente für Code, Konstanten und statische Daten. die aus den Segmenten aller Objektdateien der Bibliothek verschmolzen werden. Die dynamische Bibliothek darf auch selbst weitere dynamischen Bibliotheken referenzieren. Statt der Einsprungadresse einer ausführbare Datei enthält die dynamische Bibliothek eine Liste der Symbole, die sie exportiert.
Abbildung 3.5 zeigt schematisch eine Instanz, die drei dynamische Bibliotheken verwendet. Jedes Rechteck repräsentiert ein Segment im Speicher der Instanz. Sowohl für das Programm als auch für jede der drei Bibliotheken ist jeweils ein Segment für Code, Konstanten und statische Daten geladen. Auf eine separate Darstellung der BSS–Segmente verzichte ich hier und an anderen Stellen zugunsten der Übersichtlichkeit.

[3] findet er ein Symbol sowohl in einer statischen als auch in einer dynamischen Bibliothek... RTFM

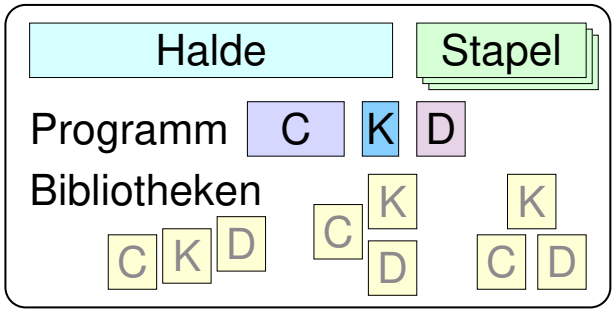


Abbildung 3.5.: Instanz mit dynamischen Bibliotheken

Stapel benötigt die Instanz entsprechend der Anzahl der Befehlsströme. Jeder Befehlsstrom kann zwischen dem Code des Programms und der Bibliotheken hin- und herspringen. Es gibt nur eine Halde, die vom Programm und allen dynamischen Bibliotheken gemeinsam verwendet wird. Die Verwaltung der Halde ist normalerweise selbst in einer dynamischen Bibliothek implementiert, die Symbole wie `malloc` und `free` exportiert.

Anmerkung: *Vereinfachend gehe ich hier davon aus, dass keine dynamischen Bibliotheken verschiedener Laufzeitumgebungen gemischt werden. Lädt man Code zum Beispiel von Python, C und Go in die gleiche Instanz, kommen mehrere Halden gleichzeitig zum Einsatz.*

Beim Start einer ausführbaren Datei sucht der dynamische Linker, ein Teil des unten beschriebenen Programmladers, nach allen direkt oder indirekt benötigten dynamischen Bibliotheken und löst die referenzierten Symbole auf. Gelingt dies nicht, scheitert der Start. Der Benutzer oder eine Administratorin kann dann den Suchpfad für dynamische Bibliotheken korrigieren oder fehlende Bibliotheken nachinstallieren.
Bibliotheken implementieren eine Programmierschnittstelle (engl.: *API, Application Programming Interface*). Mit Hilfe dynamisch gebundener Bibliotheken kann die gleiche ausführbare Datei auf unterschiedlichen Versionen oder Varianten eines Betriebssystems laufen. Die ausführbare Datei verwendet einheitliche APIs, während die installierten Bibliotheken die zum jeweiligen System passenden Implementierungen bereitstellen.

3.4. Programmlader

Der Programmlader ist eine Systemfunktion, die neue Instanzen erzeugt und darin ausführbare Dateien startet. Um geladene Segmente in mehreren Instanzen gemeinsam zu nutzen, sollte der Programmlader wissen, welche Programme und dynamischen Bibliotheken sich gerade im Speicher befinden. Den oben erwähnten dynamischen Linker betrachte ich als einen Teil des Programmladers. Zudem kann der Programmlader nach dem Ende einer Instanz aufräumen, also deren Segmente freigeben und ungenutzte dynamische Bibliotheken wieder aus dem Speicher werfen.

Das Starten einer ausführbaren Datei als Instanz umfasst folgende Schritte:

1. Segmente der ausführbaren Datei im Speicher bereitstellen:
   Code, Konstanten, statische Daten (sowohl initialisierte als auch BSS)

2. Dynamische Bibliotheken finden und deren Segmente ebenfalls bereitstellen:
   jeweils Code, Konstanten, statische Daten

3. Segmente für die Halde und den initialen Stapel bereitstellen.

4. Aufrufparameter, Umgebungsvariablen und Ähnliches im Speicher bereitstellen.

5. Den initialen Befehlsstrom der Instanz starten.

Jede Instanz erhält ihren eigenen *Adressraum*, in dem sie auf ihre Speicherinhalte, also die bereitgestellten Segmente, zugreifen kann. Code und Konstanten einer ausführbaren Datei oder einer dynamischen Bibliothek ändern sich nicht. Läuft das gleiche Programm mehrfach, oder verwenden mehrere Instanzen die gleiche dynamische Bibliothek, dann enthalten mehrere Adressräume gleiche Segmente, welche nur einmal im Speicher liegen. Für die Instanzen spielt das keine Rolle, aber es reduziert den Speicherbedarf.

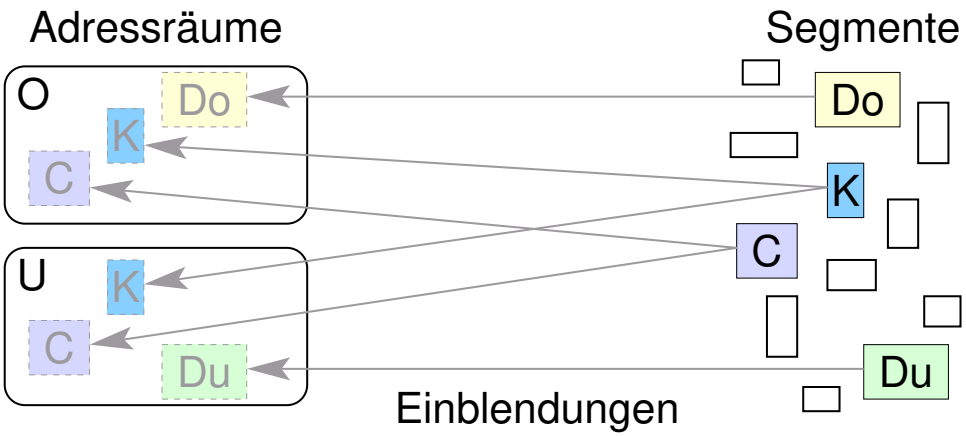


Abbildung 3.6.: Segmente in Adressräumen

Abbildung 3.6 zeigt ein Beispiel. Links sind zwei Adressräume O und U eingezeichnet, rechts ein Dutzend Segmente. Die Segmente C (Code) und K (Konstanten) liegen nur einmal im Speicher, erscheinen aber in beiden Adressräumen. Von den statischen Daten (D) braucht jede Instanz eine eigene Kopie, um ihre Werte hineinzuschreiben. Das zeigen die Segmente Do und Du, die jeweils nur in einem Adressraum erscheinen. Jeder Pfeil steht für die *Einblendung* eines bestimmten Segments in einen bestimmten Adressraum.
Der Programmlader führt Buch darüber, welche Instanzen welche Segmente aus welchen ausführbaren Dateien oder dynamischen Bibliotheken verwenden. Er nutzt die Möglichkeiten der Infrastruktur, um Adressräume, Segmente und passende Einblendungen zu erzeugen. Dynamische Bibliotheken müssen beim Start zwar gefunden, aber noch nicht in den Speicher geladen werden. Um den Startvorgang zu beschleunigen, kann man das Laden

einer Bibliothek bis zum ersten Zugriff hinausschieben. Falls eine Bibliothek bei einem Programmlauf gar nicht aufgerufen wird, entfällt das Laden komplett.
Der initiale Stapel ist theoretisch leer. Praktisch werden dort Kommandozeilenparameter und Umgebungsvariablen übergeben, also die erweiterten Argumente der Hauptfunktion `main` von C und C++:

```
int main(int argc, char *argv[], char *envp[])
```

Eine Instanz startet für gewöhnlich nicht direkt mit einer Hauptfunktion wie `main`,[4] sondern mit Initialisierungen, unter anderem für die Verwaltung der Halde. Erst danach wird die Hauptfunktion aufgerufen. Ein Rücksprung aus der Hauptfunktion, oder der Aufruf einer `exit`–Funktion, führt zu De-Initialisierungscode, der die Instanz sauber beendet. Für das Abräumen der beendeten Instanz ist wieder der Programmlader zuständig.

Neben den bisher beschriebenen ausführbaren Dateien, die von einem Linker erstellt werden und Binärcode enthalten, gibt es auch indirekt ausführbare Dateien wie Shell–Skripte. Diese brauchen einen Interpreter. Je nach Betriebssystem erkennt der Programmlader anhand der Dateiendung (`.cmd`, `.ps1`) oder des Dateiinhalts, welchen Interpreter er starten muss. Die zu interpretierende Datei wird zum Argument des Interpreters. Viele Dateiformate legen eine sogenannte *Magic Number* fest, die am Dateianfang steht.[5] Der Programmlader muss deshalb nur die ersten paar Byte des Dateiinhalts lesen, um die von ihm unterstützten Dateiformate zu erkennen. Eine besondere Rolle spielt das Hash–Bang `#!` in UNIX– und Linux–Betriebssystemen. Damit kann eine ausführbare Textdatei den für sie zuständigen Interpreter definieren, zum Beispiel:

```
#!/bin/bash
# hier steht ein bash-Skript
```

3.5. Plugins

Plugins sind Software–Module, die eine Instanz zur Laufzeit nachlädt. Technisch handelt es sich um dynamische Bibliotheken, allerdings mit Besonderheiten beim Laden und Aufrufen. Einige Praxisbeispiele für den Einsatz von Plugins:

- Webserver laden Module oder Plugins, die in ihrer Konfiguration stehen.[6]
- ODBC (engl.: *open database connectivity*) ist eine API, um Datenbanken mit SQL anzusprechen. Die ODBC–Bibliothek lädt den zur Datenbank passenden Treiber.[7]
- Medienspieler unterstützen viele verschiedene Video- und Audioformate, laden aber nur die passenden Dekoder für die gerade abzuspielenden Daten in den Speicher.

[4] *A General Overview of What Happens Before main()* , abgerufen 2022-06-30 https://embeddedartistry.com/blog/2019/04/08/a-general-overview-of-what-happens-before-main/
[5] `https://heise.de/-3120214` nennt einige Beispiele für Magic Numbers
[6] `https://httpd.apache.org/docs/2.4/developer/modguide.html`
[7] `https://www.easysoft.com/developer/interfaces/odbc/linux.html`

- Browser–Plugins gab es auch mal, aber die Netscape Plugin API (NPAPI) wurde abgeschafft.[8]
- Manche interpretierte Sprachen wie Python ermöglichen die Verwendung von C–Bibliotheken. Der Interpreter erfährt den Namen der verwendeten Bibliothek erst beim Aufruf der Ladeoperation.[9]
- Die JVM (*Java Virtual Machine*) ist ein Interpreter für Java–Bytecode. Sie unterstützte lange Zeit nur Bibliotheken für das *Java Native Interface*.[10] Das ändert sich mit der *Foreign Function & Memory API*.[11]

Beim Start einer Instanz kümmert sich der dynamische Linker des Programmladers um die Bibliotheken, welche die ausführbare Datei direkt oder indirekt braucht. Sie sind notwendig, um alle referenzierten Symbole des Programms aufzulösen. Sprünge in diese Bibliotheken erfolgen automatisch, an den Stellen, wo der Compiler die Aufrufe in den Code generiert hat.
Bei Plugins sieht das anders aus. Die Instanz erkennt, zum Beispiel an den zu verarbeitenden Daten oder über ihre Konfiguration, dass sie ein bestimmtes Plugin braucht. Sie ruft den dynamischen Linker mit dem Namen des Plugins auf. Als Ergebnis erhält sie die Symboltabelle der neu geladenen Bibliothek. Dort kann sie Funktionen nachschlagen und aufrufen. Hier sind also Funktionszeiger oder ähnliche Konzepte im Spiel. Zum Beispiel bei ODBC ist diese Fummelei in einer dynamischen Bibliothek gekapselt, die auf normalem Wege geladen wird. Schlägt das Laden eines Plugins fehl, bleibt es der Instanz überlassen, wie sie darauf reagiert. Sie könnte zum Beispiel mit eingeschränktem Funktionsumfang weiterarbeiten oder versuchen, das fehlende Plugin aus dem Internet zu ziehen. Bei dynamischen Bibliotheken, welche die ausführbare Datei benötigt, ist das nicht möglich.
Bibliotheken definieren und implementieren eine Programmierschnittstelle, eine API, die Aufrufern zur Verfügung steht. Daher der Name *Application Programming Interface*. Bei Plugins wird die Schnittstelle seitens des Aufrufers definiert, als Anforderung an jedes Plugin. Ein gängiger Name dafür ist SPI, *Service Provider Interface*. Der Unterschied wird deutlich, wenn man die Schnittstelle ändert muss. Eine neue Funktion in einer API zu ergänzen, ist kein Problem. Alte Anwendungen rufen die neue Funktion einfach nicht auf. Eine neue Funktion in ein SPI zu integrieren, ist ein Problem. Alte Plugins implementieren die neue Funktion nicht und können dadurch unbrauchbar werden.

[8] `https://support.mozilla.org/en-US/kb/npapi-plugins`
[9] `https://docs.python.org/3/library/ctypes.html`
[10] `https://docs.oracle.com/en/java/javase/17/docs/specs/jni/index.html`
[11] `https://openjdk.org/jeps/424`

3.6. Knackpunkte

- Ein Segment ist ein großer, zusammenhängender Speicherbereich mit einem bestimmten Verwendungszweck.
- Instanzen nutzen eine überschaubare Anzahl Segmente. Insbesondere:
  - Code
  - Konstanten
  - statische Daten
  - Stapel, für verschachtelte Aufrufe — ein Stapel pro Befehlsstrom
  - Halde, für Speicheranforderungen — `malloc free new delete`
- Ein Compiler übersetzt Quelltext in Objektdateien. Ein Linker kombiniert Objektdateien zu ausführbaren Dateien oder dynamischen Bibliotheken.
- Eine ausführbare Datei enthält Code, Konstanten, statische Daten sowie eine Einsprungadresse. Sie kann Symbole dynamischer Bibliotheken referenzieren.
- Eine dynamische Bibliothek enthält Code, Konstanten, statische Daten sowie exportierte Symbole. Sie kann Symbole anderer dynamischer Bibliotheken referenzieren.
- Der Programmlader erstellt Instanzen aus ausführbaren Dateien und dynamischen Bibliotheken. Sein dynamischer Linker löst alle referenzierten Symbole auf.
- Jede Instanz hat einen eigenen Adressraum, in den ihre Segmente eingeblendet sind. Der Programmlader erstellt und initialisiert die für den Start benötigten Segmente und Einblendungen.
- Die Instanz erhält beim Start Aufrufargumente und Umgebungsinformationen.
- Der Programmlader startet den ersten Befehlsstrom der Instanz, an der Einsprungadresse aus der ausführbaren Datei.
- Der Programmlader führt Buch über Code– und Konstanten–Segmente, die von mehreren Instanzen gemeinsam genutzt werden können. Das spart Speicher.
- Plugins sind dynamische Bibliotheken, die eine Instanz explizit nachlädt.

4. Ein-/Ausgabe

Datenströme (engl.: *streams*) haben sich als Abstraktion für die Kommunikation durchgesetzt. Sie verbinden eine Instanz mit den verschiedensten Datenquellen und -senken, welche meist in der Außenwelt liegen. Die jeweilige API findet man in den Standardbibliotheken der verbreiteten Programmiersprachen. In Beispielen verwende ich die Funktionsnamen der C–API, gelegentlich ergänzt um Java–Klassen. Da die Instanz über eine standardisierte API auf Datenströme zugreift, spielt es für sie keine Rolle, ob die Gegenseite der Verbindung im Kern, in einem anderen Adressraum, in der Peripherie oder auf einem anderen Rechner liegt. Eine Instanz kann auch problemlos mit mehreren Strömen hantieren, die das Betriebssystem auf unterschiedliche Art implementiert.

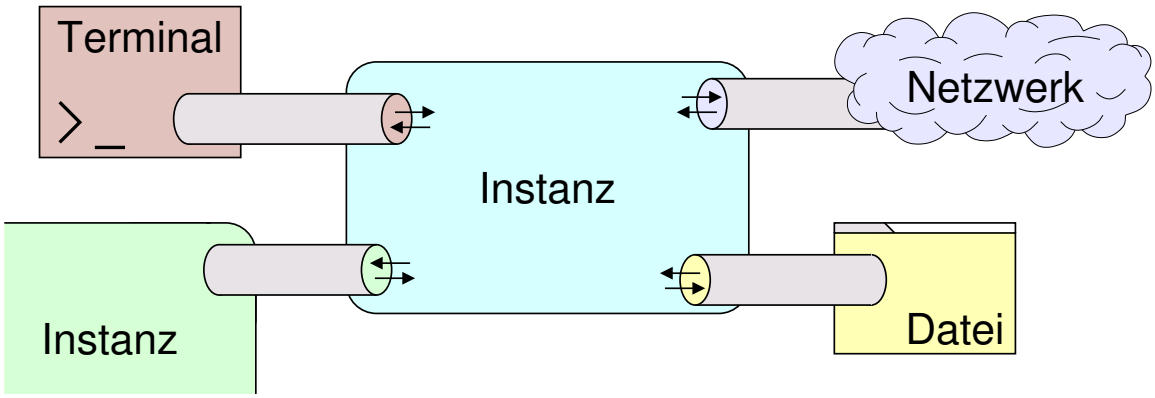


Abbildung 4.1.: Instanz mit Datenströmen

Abschnitt 4.1 beschreibt die abstrakte Sicht auf Datenströme. Anschließend erklärt Abschnitt 4.2 die Pufferung, ohne die es mit der Geschwindigkeit der Kommunikation mau aussähe. Schließlich geht Abschnitt 4.3 auf die Besonderheiten einiger ausgewählter Datenquellen und -senken ein. Dabei treten auch die Grenzen der Abstraktion zu Tage.

4.1. Datenströme

Ein Datenstrom verbindet eine Instanz mit einer Datenquelle zum Lesen oder mit einer Datensenke zum Schreiben oder beides. Im letzten Fall kann man die Verbindung als zwei getrennte Ströme betrachten, einen Eingabestrom und einen Ausgabestrom. Abbildung 4.1 zeigt Verbindungen als Rohre zu verschiedenen Komponenten außerhalb des Adressraums der Instanz. Die Analogie krankt daran, dass ein Rohr nicht in beiden Richtungen gleichzeitig (engl.: *full duplex*) funktioniert.

Ein Eingabestrom (engl.: *input stream*) liefert Daten als eine Folge von Bytes. Die lesende Instanz kann ein einzelnes Byte abholen oder gleich einen ganzen Block von Bytes auf einmal fordern. Ob sie den auch erhält, hängt von den Umständen ab. Ein Datenstrom kann ins Stocken geraten, zum Beispiel weil Daten von der Festplatte oder von einem anderen Rechner nicht schnell genug ankommen. Typischerweise arbeiten Leseoperationen blockierend. Beim Lesen eines einzelnen Byte blockiert die Instanz, bis sie eines erhält. Beim Lesen eines Blocks erhält sie nur so viele Bytes, wie gerade verfügbar sind. Ist überhaupt nichts verfügbar, blockiert sie bis zum Eintreffen mindestens eines Bytes. Ein Datenstrom kann enden, zum Beispiel weil er das Ende einer Datei erreicht oder weil die Gegenseite auf einem anderen Rechner die Verbindung schließt. Die Leseoperationen zeigen das Stromende durch spezielle Rückgabewerte an. Man spricht von EOF (engl.: *end of file*), selbst wenn der Strom nicht aus einer Datei gespeist wird.

Ein Ausgabestrom (engl.: *output stream*) akzeptiert Daten als eine Folge von Bytes. Die schreibende Instanz kann ein einzelnes Byte übergeben oder gleich einen ganzen Block von Bytes. Die Schreiboperation blockiert, bis die Daten in einen Puffer übernommen, auf den Weg gebracht oder auf der Gegenseite angekommen sind. Die Details hängen vom Strom ab. In einigen Fällen, zum Beispiel bei Dateizugriffen, legt die Instanz selbst fest, wie das System mit den geschriebenen Daten umgeht.

Die Außenwelt entzieht sich der direkten Kontrolle durch die Instanz. Deshalb ist bei jeder Operation mit Fehlern zu rechnen. Festplatten können kaputt gehen, Netzwerkverbindungen abbrechen, Dateien während des Zugriffs gelöscht werden. Zumindest sollte eine Anwendung Fehler bei der Ein- oder Ausgabe erkennen und mit einer Meldung an den Benutzer quittieren. Leider arbeiten viele Entwickler in diesem Punkt schlampig. In C wird der Rückgabewert vieler Operationen, nämlich eine Fehlernummer, oft ignoriert. In Java findet man `catch`-Blöcke, die eine IOException fangen, aber dann die Ausführung des Programms fortsetzen, als sei nichts passiert. Dass Daten nicht gesichert wurden erkennt der Benutzer später, wenn er wieder darauf zugreifen will.

Datenströme bieten keine Hilfe bei der Interpretation des Inhalts. Eine lesende Instanz muss selbst wissen und prüfen, wie die eintreffenden Daten aussehen. Arbeitet sie zum Beispiel mit Bildern, kann sie anhand der „Magic Number“ am Anfang des Datenstroms deren Binärformat erkennen. Arbeitet sie mit Texten, muss der verwendete Zeichensatz vielleicht als Parameter übergeben oder konfiguriert werden.

Aus dem Unix–Umfeld mit der Programmiersprache C stammt die Konvention, dass einer Anwendung schon beim Start drei Datenströme zur Verfügung stehen: Standardeingabe (`stdin`), Standardausgabe (`stdout`) und Standardfehlerausgabe (`stderr`). Die Flexibilität von Kommandozeileninterpretern beruht mit darauf, dass man Daten durch mehrere Verarbeitungsschritte filtern kann, indem man die Standardaus- und -eingabeströme verschiedener Anwendungen miteinander verkettet.

4.2. Pufferung

Jeder Zugriff auf einen Datenstrom erfordert einen Kernaufruf. Im Vergleich zu Funktionsaufrufen im eigenen Adressraum sind Kernaufrufe sehr langsam, da mindestens der Umgebungswechsel von der Instanz zur Unterbrechungsbehandlung und wieder zurück anfällt. Wenn der Datenstrom nicht im Kern, sondern in einer anderen Instanz implementiert ist, erhöht sich der Aufwand noch einmal. Und wenn ein Peripheriezugriff notwendig ist, noch mehr. Mit anderen Worten, das Lesen aus oder Schreiben in einen Datenstrom ist eine teure Operation. Man sollte besser große Blöcke lesen und schreiben als einzelne Byte, um die Anzahl der teuren Aufrufe zu reduzieren.
Die Pufferung gehört zur API für Datenströme. Sie ist direkt in der Bibliothek implementiert, arbeitet also im Adressraum der Instanz. Für jeden gepufferten Strom verwaltet die Bibliothek einen eigenen Puffer. Dessen Größe kann die Instanz einstellen, zum Beispiel auf 4096 Byte. Beim ersten Lesezugriff ist der Puffer leer. Die Bibliothek ruft die teure Leseoperation des Datenstroms auf, um den Puffer zu füllen. Die folgenden Lesezugriffe bedient sie direkt aus dem Puffer, bis dieser geleert ist. Beim Schreiben läuft es umgekehrt. Die Bibliothek legt geschriebene Daten erst einmal im Puffer ab. Wenn der Puffer voll ist oder die Instanz `flush` (dt.: leeren) aufruft, wandert der Inhalt des Puffers mit nur einer teuren Schreiboperation in den Datenstrom. Das Schließen des gepufferten Datenstroms beinhaltet ebenfalls ein `flush`.
Eine Instanz, die in einen gepufferten Strom schreibt, muss an sinnvollen Stellen im Code `flush` aufrufen. Das ist vor allem dann wichtig, wenn jemand auf die geschriebenen Daten wartet. Zum Beispiel ein Benutzer, der etwas auf dem Bildschirm sehen will. Oder ein Administrator, der die Einträge in einer Logdatei mitverfolgt. Oder ein Webserver, der die HTTP–Anfrage erwartet, bevor er Daten liefert. Bei textbasierter Ausgabe ist es üblich, jeweils am Zeilenende ein `flush` durchzuführen. Wenn eine Anwendung aber einzelne Zeichen als Fortschrittsanzeige ausgibt, muss sie das nach jedem Zeichen tun. Bei der HTTP–Anfrage ergibt ein `flush` dagegen erst am Ende der kompletten Anfrage Sinn, obwohl sie aus Textzeilen besteht. Hier ist der Entwickler gefordert, sinnvolle Abschnitte im Datenstrom zu definieren.

Operation	**gepuffert**	**direkt**
Datei öffnen	`fopen`	`open`
Daten lesen	`fread`	`read`
Daten schreiben	`fwrite`	`write`
Strom schließen	`fclose`	`close`

Abbildung 4.2.: C–Funktionen für Datenströme

Abbildung 4.2 nennt die wichtigsten C–Funktionen für Datenströme. Als Beispiel für das Öffnen eines Stroms habe ich das Öffnen einer Datei gewählt. Die oft zitierte Unix–Philosophie „everything is a file“ bezieht sich darauf, dass man auch viele andere Arten von Strömen so öffnen kann. Für jede Operation gibt es eine gepufferte und eine ungepufferte Variante. Der Programmierer entscheidet sich für eine der Spalten und ruft dann entweder nur die gepufferten oder nur die ungepufferten Funktionen. Zur Identifikation des Daten-

stroms dient im gepufferten Fall ein Zeiger auf eine Datenstruktur (`FILE*`), im ungepufferten Fall eine Nummer.
In Java repräsentieren `InputStream` bzw. `OutputStream` aus `java.io` beliebige binäre Datenströme. Für textbasierte Ströme verwendet man `Reader` bzw. `Writer`, die automatisch zwischen Zeichen und Bytes konvertieren. Für Dateizugriffe gibt es von allen vier abstrakten Klassen eine Implementierung, deren Name mit `File` beginnt. Diese Implementierungen arbeiten ungepuffert. Für gepufferte Ströme gibt es ebenfalls von allen vier abstrakten Klassen eine Implementierung, deren Name mit `Buffered` beginnt. Diese erwarten jeweils einen passenden Strom als Konstruktorargument. Man öffnet also zuerst einen ungepufferten Datenstrom und legt dann einen Puffer dafür an. Anschließend verwendet man nur noch den gepufferten Strom.

Anmerkung: *Hier geht es nur um die Pufferung bei der Instanz, in ihrem eigenen Adressraum. Der Datenstrom selbst kann ebenfalls Puffer enthalten. Zum Beispiel um Netzwerkprotokolle wie TCP zu implementieren. Oder weil eine Festplatte Daten in Blöcken anliefert und erwartet. Der richtige Umgang mit solchen Puffern ist jeweils ein eigenes Thema.*

4.3. Besonderheiten

Solange man einen vorhandenen Datenstrom rein sequentiell nutzt, also entweder nur daraus liest oder nur hinein schreibt, muss man sich nicht viele Gedanken machen. Das ändert sich, sobald eine Anwendung spezielle Funktionen nutzen will, die nur von bestimmten Arten von Strömen bzw. Gegenseiten in der Außenwelt unterstützt werden. Außerdem gibt es Unterschiede beim Öffnen verschiedener Datenströme. Die folgenden Beispiele habe ich ausgewählt, um die wohl häufigsten Fälle abzudecken. Es soll aber keine erschöpfende Aufzählung sein.

4.3.1. Pipes

Eine *Pipe* (dt.: Leitung, Röhre), auch FIFO genannt (engl.: *first in, first out*), verbindet einen Ausgabestrom mit einem Eingabestrom auf dem gleichen Rechner. Auf diesem Weg schickt eine Instanz Daten an eine andere. Eine Übertragung in umgekehrter Richtung ist nicht vorgesehen. Das Betriebssystem übernimmt die Pufferung der übertragenen Daten in begrenztem Umfang. Ist der Puffer voll, blockiert die schreibende Instanz. Im Prinzip können sowohl Ausgabestrom als auch Eingabestrom von mehreren Instanzen genutzt werden. Allerdings kommen die Daten dann sehr leicht durcheinander.
Eine Pipe ist ein Datenstrom in Reinkultur, eine direkte Verbindung zwischen Instanzen, ohne Umweg über Peripherie oder Netzwerk. Im Vergleich zu den Kanälen aus Kapitel 2 verschwinden bei einer Pipe die Nachrichtengrenzen. Der Empfänger holt sich die Daten in beliebigen Portionsgrößen. Soll der Inhalt trotzdem aus einer Folge von Nachrichten bestehen, müssen deren Grenzen aus den Daten ersichtlich werden.
In der Kommandozeile von Linux, Unix oder Windows zeigt das Pipe–Symbol `|` an, dass die Standardausgabe des davor stehenden Kommandos mit der Standardeingabe des nachfolgenden Kommandos verknüpft werden soll. Das funktioniert mit einer anonymen Pipe,

die nur im Speicher existiert. Die POSIX–Funktion zum Erzeugen einer anonymen Pipe heißt `pipe`. Linux kennt außerdem `pipe2` mit zusätzlichen Flags, um das Verhalten der Pipe zu beeinflussen. Benannte Pipes (engl.: *named pipes*) erzeugt man mit `mkfifo`. Sie sind über einen Dateinamen zu finden, obwohl die übertragenen Daten im Speicher bleiben. Auf diesem Weg können auch unabhängig voneinander startende Instanzen eine Verbindung miteinander aufbauen.
Das Verknüpfen von einfachen Kommandos über Pipes ist ein Grundprinzip der Unix–Shell, aber nicht ohne Kritik.[1] Da die Verbindung nur den Datenstrom überträgt, muss das empfangende Kommando diese Daten neu parsen. Es hat keinen Zugriff auf Metadaten, die das sendende Kommando vielleicht schon gesammelt hat, aber nicht in den Datenstrom kodiert. Die Windows PowerShell[2] geht einen anderen Weg und übergibt typisierte Objekte zwischen Kommandos. Allerdings sind ihre Kommandos keine eigenen Instanzen, sondern laufen im gleichen Adressraum.

4.3.2. Terminals

Abbildung 4.3.: Terminal

Quelle: Bilby, über Wikimedia Commons (Basic_Four_terminal.jpg)
Lizenz: CC BY 3.0

Ein Terminal oder eine Konsole ist ein Ein- und Ausgabegerät für Text. Früher handelte es sich dabei noch um Hardware, eine Kombination aus Anzeige und Tastatur wie in Abbildung 4.3. Aktuelle Betriebssysteme emulieren das Verhalten von Terminals für die Bedienung aus der Kommandozeile. Die Emulation läuft in einem Fenster der graphischen Benutzeroberfläche oder als virtuelle Konsole ohne graphische Oberfläche. Auch der Zugriff von einem anderen Rechner aus, zum Beispiel mit `ssh`, verwendet eine Terminal–Emulation.
Eine interaktive Shell greift auf das Terminal über die Datenströme einer bidirektionalen Verbindung zu. In den Ausgabestrom schreibt sie Texte zur Anzeige, aber auch Steuercodes, sogenannte Escape–Sequenzen. Mit den Steuercodes kann sie unter anderem den Cursor positionieren und verschiedene Farben zur Darstellung der Texte wählen. Eingaben des Benutzers erhält die Shell über den Eingabestrom. Das können getippte Buchstaben sein, aber auch Steuercodes von Funktions- oder Cursortasten.

[1] „10 Things I Hate About (U)NIX“, Seite 2 und Seite 10, abgerufen 2017–02–12
`https://www.informit.com/articles/article.aspx?p=424451&seqNum=2`
[2] „Scripting with Windows PowerShell“, abgerufen 2017–02–12
`https://technet.microsoft.com/en-us/library/bb978526.aspx`

Führt die Shell ein einzelnes Kommando aus, dann übernimmt dieses die Ansteuerung des Terminals. Texteditoren wie `nano`, `vim` oder `emacs` (mit der Option `-nw`) bieten auf diesem Weg eine interaktive Benutzerschnittstelle. Das Programm `ls` nutzt die Möglichkeiten, um Verzeichniseinträge wie Dateien, Unterverzeichnisse und symbolische Links mit unterschiedlichen Farben in mehreren Spalten anzuzeigen. Verwendet man es allerdings in einer Pipe, ändert es sein Verhalten. Jeder Eintrag erscheint in einer eigenen Zeile, Steuercodes zum Wechseln der Farbe fehlen ganz. Ein interaktiver Editor verweigert die Arbeit komplett, wenn er keinen Zugriff auf eine Terminal–Emulation hat.
Jenseits der Möglichkeiten reiner Datenströme muss eine Anwendung also auch erkennen können, ob hinter der Standardein- und -ausgabe ein Terminal steckt oder nicht. Dazu kommen Abfragen, insbesondere für die Anzahl der Zeilen und Spalten. Programme, die mit den zusätzlichen Möglichkeiten eines Terminals nichts anzufangen wissen, sparen sich die Abfragen. Sie verwenden ihre Datenströme rein sequentiell und funktionieren gleichermaßen im Terminal oder in einer Pipe.
Zeichenorientierte Terminals unterstützen verschiedene Modi, zwischen denen die Anwendung mit Steuercodes umschaltet. Im gepufferten Modus editiert der Benutzer eine Zeile direkt im Terminal. Eingegebene Zeichen zeigt das Terminal selbständig an, die Löschtaste und horizontale Cursorbewegungen funktionieren. Die Eingabetaste schickt eine komplette Zeile an die Anwendung. Dieses Verhalten eignet sich gut für die Bedienung einer Shell. Im ungepufferten Modus schickt das Terminal jeden Tastendruck an die Anwendung, ohne selbst etwas an der Anzeige zu ändern. Damit können Editoren die Cursorposition steuern und spezielle Aktionen als Reaktion auf Tastendrücke durchführen. Im Gegenzug muss ein Editor aber jedes eingegebene Zeichen selbst wieder ausgeben, damit es in der Anzeige erscheint. Auch die Löschtaste und andere Sonderfunktionen muss der Editor selbst auswerten. Soll der Benutzer ein Passwort eingeben, das nicht im Terminal angezeigt werden darf, kommt ebenfalls der ungepufferte Modus zum Einsatz.

Anmerkung: *Der Modus eines Terminals hat erst einmal nichts mit der Pufferung von Datenströmen auf der Anwendungsseite zu tun. Allerdings ergibt es wenig Sinn, wenn eine Anwendung ihr Terminal im ungepufferten Modus betreibt, aber selbst zu puffern versucht.*

Historisch bedingt verwendeten die Terminals verschiedener Hersteller unterschiedliche Escape–Sequenzen. Eine Anwendung muss also nicht nur erkennen, dass sie mit einem Terminal verbunden ist. Sie muss außerdem noch wissen, welchen Satz von Escape–Sequenzen das Terminal oder die Terminal–Emulation unterstützt. Diese Information, zum Beispiel „VT100“ oder „xterm“, steht in der Umgebungsvariablen `TERM`. Einige Bibliotheken helfen beim Ansteuern von Terminals. `terminfo` enthält eine Datenbank von Steuercodes für die gängigen Terminals und Emulationen. `ncurses` bietet Funktionen zum Darstellen von Benutzerelementen wie Menüs und Dialogen in einem Terminal.
Neben den beschriebenen, zeichenorientierten Terminals gibt es im Mainframe– und Großrechner–Umfeld auch blockorientierte Terminals. Diese erhalten von der Anwendung eine Bildschirmmaske mit darzustellenden Inhalten und definierten Eingabefeldern. Das Terminal übernimmt die komplette Cursorsteuerung und erlaubt es dem Benutzer, alle Eingabefelder der Maske auszufüllen. Auf Knopfdruck schickt es dann die gesammelten Eingaben in einem Block zurück an die Anwendung.

4.3.3. Dateien

Eine Datei ist ein Datenlager außerhalb der zugreifenden Instanzen. Sie besteht aus einer Folge von Bytes. Inhalt und Länge der Datei ändern sich durch Schreiboperationen. Mit Ausnahme von temporären Dateien ist die Existenz einer Datei nicht an die Existenz der Instanzen gebunden, die darauf zugreifen. Dateien können schon vor dem Programmstart Eingabedaten enthalten und überdauern das Programmende. Liegt eine Datei auf einem persistenten Datenträger, überlebt sie auch einen Neustart des Rechners.
Eine Instanz verbindet sich mit einer Datei, indem sie sie öffnet. Die Datei wird durch einen Namen identifiziert. Eine Ausnahme bilden wiederum temporäre Dateien, die auch anonym sein können. Je nach Modus entsteht beim Öffnen ein Eingabe- oder ein Ausgabestrom oder beides. Um den Inhalt einer Datei als Eingabestrom bereitzustellen, muss sich das Betriebssystem merken, bis zu welcher *Position* die Datei schon gelesen wurde. Das Setzen der Position ermöglicht Zugriffsmuster auf Dateien, die über sequentielle Ströme hinausreichen. Um diese Zugriffsmuster geht es im Folgenden.

Sequentiell

Das Öffnen einer Datei zum Lesen erzeugt einen Eingabestrom. Dieser steht zunächst an der Position 0, also am Anfang der Datei. Beim Lesen liefert er nach und nach deren kompletten Inhalt. Am Dateiende meldet der Eingabestrom diese spezielle Situation, durch einen leeren Block bzw. den Wert EOF (engl.: *end of file*). Der Strom bleibt aber offen und an seiner letzten Position. Anschließende Leseoperationen können erfolgreich sein, falls eine andere Instanz neue Daten anhängt. Sequentielles Lesen ist der Normalfall, um eine Datei vollständig zu laden. Beim Abspielen von Musikdateien kann man diese ebenfalls sequentiell lesen, ohne sie vollständig im Speicher zu halten. Zum Anzeigen von Logdateien ist es praktisch, den Strom geöffnet zu halten und regelmäßig auf neu angehängte Daten zu prüfen.
Das Anlegen einer neuen Datei zum Schreiben erzeugt einen Ausgabestrom. Der steht gezwungenermaßen an Position 0 der Datei, weil diese zunächst leer ist. Jede Schreiboperation hängt Daten an und vergrößert die Datei. Die Position des Ausgabestroms wandert mit und bleibt immer am Dateiende. Beim Öffnen einer existierenden Datei zum Schreiben bestimmt die Instanz, ob sie neue Daten anhängen oder die alte Datei überschreiben will. Im ersten Fall wird der Ausgabestrom ans Ende der Datei positioniert. Im zweiten Fall löscht das System den Inhalt der Datei und setzt den Strom an Position 0, was ebenfalls dem Ende der Datei entspricht. Anhängen ist bei Logdateien sinnvoll. Überschreiben ist der Normalfall beim Sichern von Dateien aus einem Editor, einer Bildbearbeitung oder anderen Programmen mit ähnlicher Nutzung. Aufnahmen von Audio- oder Videoströmen landen meist in neuen Dateien.
Natürlich ist es auch möglich, zum Schreiben zu öffnen und den Strom an den Anfang zu positionieren, ohne den Inhalt der Datei zu löschen. In Kombination mit sequentiellem Schreiben fällt mir dafür aber keine sinnvolle Anwendung ein, deshalb folgt das erst im nächsten Abschnitt. Gleiches gilt für das Öffnen in beide Richtungen, zum Lesen und Schreiben.

Beliebig

Durch Setzen der Position des Eingabestroms kann eine Instanz beliebige Teile einer Datei in beliebiger Reihenfolge (engl.: *random access*) beliebig oft lesen, und uninteressante Bereiche überspringen. Zum Beispiel lädt man beim Start eines Programms nur die Segmente, die dafür notwendig sind. Ausführbare Dateiformate sind darauf ausgelegt. Schon zuvor holt sich der Linker beim Erzeugen von ausführbaren Dateien nur die benötigten Objekte aus statisch gebundenen Bibliotheken. Beim Abspielen von Videos lässt Vor- oder Zurückspulen die Dateiposition springen.
Die Operation zum Setzen der Position heißt *seek* (dt.: aufsuchen), die POSIX–Funktionen sind `fseek` für gepufferte und `lseek` für ungepufferte Ströme. Positionsangaben erfolgen in Byte, entweder absolut oder relativ zur aktuellen Position oder zum Dateiende. Abbildung 4.4 veranschaulicht das an einem Beispiel. Die Zeichen innerhalb des Balkens sind der Inhalt der Datei. Zahlen darunter markieren verschiedene absolute Positionen, links vom Dateianfang, rechts vom Dateiende aus gezählt. Der rote Pfeil steht für eine aktuelle Position, welche sich bei Zugriffen ändert.

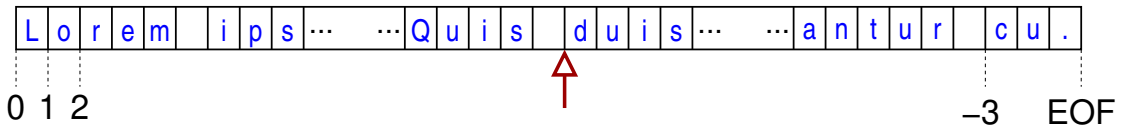


Abbildung 4.4.: Dateiposition

Auch beim Schreiben in den Ausgabestrom bestimmt die aktuelle Position, wo in der Datei die Daten landen. Beispiele für reines Schreiben mit Sprüngen sind eher theoretisch. Manche Datenstrukturen enthalten Längenangaben, deren Werte man erst kennt, wenn die nachfolgenden Teile der Struktur aufgebaut sind. Diese Längen könnte man beim Schreiben erst auf Null setzen, um später die Position wieder anzuspringen und den richtigen Wert nachzutragen. Allerdings ist es bequemer, eine solche Datenstruktur erst einmal im Speicher aufzubauen und dann sequentiell in die Datei zu schreiben. Oder man wählt gleich eine Datenstruktur, die sich besser zum sequentiellen Schreiben eignet.
Öffnet man eine Datei zum Lesen und Schreiben, hat man sowohl einen Eingabe- als auch einen Ausgabestrom. Da sie zur gleichen Datei führen, sind diese Ströme nicht unabhängig. Sie teilen sich sowohl die Position in der Datei als auch den Puffer im Adressraum der Instanz. Außerdem erwartet jeder Entwickler, beim Lesen die Daten zu erhalten, die er vorher in die Datei geschrieben hat. Das ist allerdings keineswegs selbstverständlich. Für gepufferte Ströme verlangt POSIX beim Wechsel zwischen Schreiben und Lesen erst eine Positionierung mit `fseek` oder `fgetpos`. Die Position muss sich dabei nicht einmal ändern. Der Aufruf leert den Puffer und erzwingt die Synchronisation zwischen Ein- und Ausgabestrom. Für ungepufferte Dateizugriffe spezifiziert POSIX, dass `read` nach `write` die geschriebenen Daten liefert. Die Linux–Manpage zu `write` ergänzt trocken:

Note that not all file systems are POSIX conforming.[3]

[3] `https://linux.die.net/man/2/write`, abgerufen 2017–02–12

Blockweise

Bei Datenbank–ähnlichen Anwendungen kommen Dateien vor, deren Inhalt logisch in Blöcke (engl.: *records*) fester Länge aufgeteilt ist. Die Anwendung liest und schreibt nur ganze Blöcke. Durch Multiplikation der Blockgröße mit der Blocknummer bestimmt sie die Position in der Datei. Bis auf die freiwillige Einschränkung der Position und des Umfangs beim Lesen und Schreiben entspricht das dem Zugriffsmuster „Beliebig“ von oben. Gleichzeitig ist es das einzige sinnvolle Beispiel für kombiniertes Lesen und Schreiben, das mir einfällt.
Tatsächlich gibt es Dateisysteme, die speziell auf die Verarbeitung blockorientierter Daten ausgelegt sind (engl.: *record–oriented file system*). Sie betrachten eine Datei nicht als Folge von Bytes, sondern als Folge von Records. Dabei dürfen die Records einer Datei sogar variable Längen haben. Solche Dateisysteme findet man einerseits bei Mainframes und Großrechnern, andererseits auch im Kleinen bei Smart Cards. Der Zugriff auf den Inhalt erfolgt dann aber nicht über Datenströme, sondern mit spezialisierten APIs. Diese unterstützen teilweise auch erweiterte Funktionen, zum Beispiel den Zugriff auf Blöcke über einen Schlüsselwert (engl.: *key*) statt der Position. Ein solches Dateisystem nimmt also bereits Grundzüge einer Datenbank an.

4.3.4. Sockets

Mit Sockets baut man unter anderem Verbindungen zwischen Client– und Server–Instanzen auf. Das funktioniert lokal, aber auch über Netzwerke. Ein Server erzeugt einen Server–Socket mit einer bestimmten Adresse. Im lokalen Fall ist die Adresse ein Dateiname, übers Netzwerk meist eine IP–Adresse mit Portnummer. Ein Client erzeugt einen passenden Socket, um sich mit dem Server zu verbinden. Der Server entscheidet, wann er eine Verbindung akzeptiert. Danach besteht je ein Datenstrom pro Richtung. Im lokalen Fall ist das ein Vorteil gegenüber Pipes, die nur in eine Richtung funktionieren. Ähnlich `pipe` erzeugt `socketpair` zwei lokale, miteinander verbundene Sockets, also eine bidirektionale Pipe.

Anmerkung: *Client und Server sind hier nur die Rollen beim Verbindungsaufbau. Daraus folgt nicht, dass die beiden Instanzen anschließend auch nach diesem Interaktionsmuster miteinander kommunizieren.*

Das verbreitetste Protokoll für Verbindungen über Netzwerke ist TCP (engl.: *Transmission Control Protokoll*). Es überträgt Daten paketweise und kümmert sich darum, dass sie auf der Gegenseite in der richtigen Reihenfolge wieder zusammengesetzt werden. Dazu verwaltet es auf beiden Seiten Puffer für beide Richtungen. Empfangene Pakete bringt es dort in die richtige Reihenfolge. Gesendete Daten bleiben bis zur Bestätigung des Empfangs gepuffert, falls Pakete unterwegs verloren gehen. Zusätzlich verwenden beide Instanzen meist noch Puffer im eigenen Adressraum, wie in Abschnitt 4.2 beschrieben.
Das Schließen der Verbindung leert sämtliche Puffer. TCP erlaubt es, die Ströme in beiden Richtungen unabhängig zu schließen. So kann eine Instanz durch das Schließen des Ausgabestroms anzeigen, dass sie keine weiteren Daten mehr schickt. Trotzdem kann sie noch eine Antwort von der Gegenseite lesen. Erst wenn beide Richtungen auf beiden Seiten geschlossen sind, ist die TCP–Verbindung vollständig beendet.

Um die Kommunikation zu verschlüsseln, kommt zum Beispiel das Protokoll TLS (engl.: *Transport Layer Security*) zum Einsatz, welches das veraltete SSL (engl.: *Secure Socket Layer*) abgelöst hat. Auch TLS implementiert bidirektionale Verbindungen und nutzt für die Übertragung Sockets. Es hängt aber von der verwendeten Programmiersprache ab, ob man die verschlüsselten Verbindungen mit der API für Datenströme nutzen kann. Implementiert wird TLS von Bibliotheken, die in den Adressräumen der beiden kommunizierenden Instanzen arbeiten. In C ruft man spezielle Lese- und Schreiboperationen der Bibliothek auf, zum Beispiel `SSL_read` und `SSL_write` bei OpenSSL.[4] In objektorientierten Programmiersprachen wie Java kann die Bibliothek nach dem Verbindungsaufbau Datenströme mit der üblichen API bereitstellen.

[4] `https://www.openssl.org/docs/man1.1.1/man3/SSL_read.html`, abgerufen 2020-02-27

5. Dateipfade

Eine wichtige Systemfunktion ist der Zugriff auf Dateien und Verzeichnisse. Pfadnamen identifizieren Dateien und Verzeichnisse. Die Systemfunktion löst Pfadnamen zu konkreten Objekten in einem Dateisystem auf. In diesem Kapitel betrachten wir die Systemfunktion in erster Linie aus der Nutzersicht. Administrative Aspekte folgen später in Kapitel 17. Zuletzt stellt Kapitel 18 die Verbindung her. Der Begriff *Dateisystem* spielt in allen drei Kapiteln eine zentrale Rolle. Er trägt mehrere Bedeutungen, die ich unten kurz erläutere. Auf die Implementierung von Dateisystemen gehe ich nicht ein, der Detailgrad würde den Rahmen der Vorlesung sprengen.
Ein Dateisystem organisiert die Ablage von Daten auf einem persistenten Speichermedium: Festplatte, SSD, CD–ROM, USB–Stick, SD–Karte und viele mehr. Dateien sind Datencontainer unterschiedlicher Größe, die Daten in einem bestimmten Format enthalten. Dateisysteme verwalten die Container, unabhängig von den darin verwendeten Dateiformaten. Dateien tragen Namen und liegen in hierarchischen Verzeichnisstrukturen. Über Pfade, die aus Verzeichnis- und Dateinamen zusammengesetzt sind, greift man auf bestimmte Dateien zu. Instanzen öffnen Ein-/Ausgabeströme zu Dateien, wie in Abschnitt 4.3.3 beschrieben.

Obacht: *Alle Aussagen im vorigen Abschnitt gelten nur eingeschränkt. Es gibt flüchtige Speichermedien, mehrere Container in einer Datei, leere Dateien, namenlose Dateien, Verzeichnisse ohne Hierarchie, Zugriffe ohne Ein-/Ausgabeströme, und vieles mehr.*

Die Systemfunktion ermöglicht den Zugriff auf Inhalte von Dateisystemen angeschlossener oder anderweitig erreichbarer Speichermedien. Wie erwähnt trägt der Begriff *Dateisystem* mehrere Bedeutungen. Spricht man vom Zugriff auf *das* Dateisystem, ist meist die Systemfunktion gemeint. Ext4,[1] NTFS,[2] APFS[3] und FAT32[4] sind Namen verschiedener Ablageformate, die auf Speichermedien zum Einsatz kommen. Diese Formate, ebenso wie ihre Implementierungen, bezeichnet man als Dateisysteme. Nicht zuletzt bilden die auf einem Speichermedium in einem bestimmten Dateisystemformat abgelegten Dateien und Verzeichnisse ein konkretes Dateisystem.
Abschnitt 5.1 erklärt, wie das Betriebssystem Pfade auflöst, um ein bestimmtes Objekt in einem Dateisystem zu finden. Abschnitt 5.2 führt symbolische Links ein, die Pfade umbiegen. Abschnitt 5.3 beschreibt den Umgang mit mehreren Dateisystemen, insbesondere das Einhängen oder *Mounten* eines Dateisystems. Abschnitt 5.4 betrachtet verschiedene Operationen auf Verzeichniseinträgen. Abschnitt 5.5 fasst die wichtigsten Punkte dieses Kapitels zusammen.

[1] `https://opensource.com/article/18/4/ext4-filesystem`
[2] `https://www.ntfs.com/`
[3] `https://developer.apple.com/documentation/foundation/file_system`
[4] `https://wiki.osdev.org/FAT`

5.1. Pfadauflösung

Ein Pfad ist eine Zeichenkette, deren Komponenten durch / oder \ getrennt sind. Die vorne stehenden Komponenten bezeichnen Verzeichnisse oder ähnliche Dateisystemobjekte, die letzte Komponente eine Datei, ein Verzeichnis oder ein ähnliches Objekt. In manchen Betriebssystemen kann ganze vorne ein Laufwerksbuchstabe oder Gerätename stehen, durch Doppelpunkt abgetrennt. In den Beispielen verwende ich / als Trennzeichen, sofern es nicht explizit um Windows geht. Zunächst betrachten wir ein einzelnes, konkretes Dateisystem, als sei es das einzig zugreifbare.

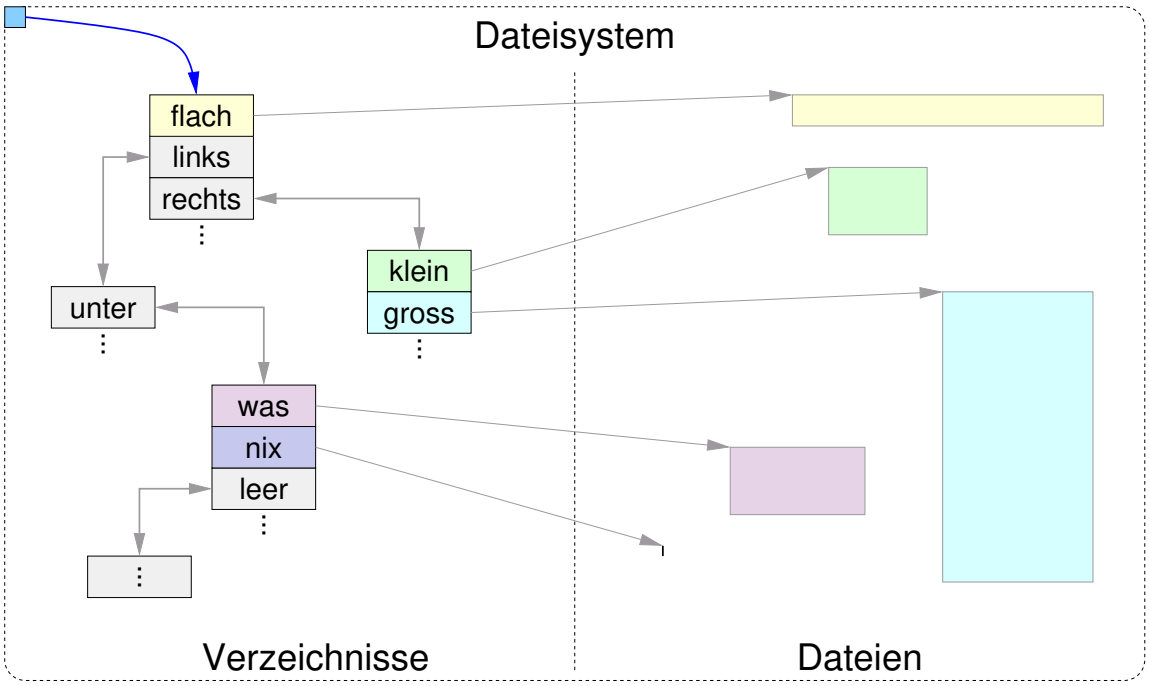


Abbildung 5.1.: Verzeichnisse und Dateien

Abbildung 5.1 zeigt Verzeichnisse auf der linken Seite, Dateien rechts. Verzeichnisse sind Tabellen, welche das Dateisystem interpretiert. Jeder Verzeichniseintrag hat einen Namen und verweist auf ein Dateisystemobjekt, im Beispiel entweder auf ein anderes Verzeichnis oder auf eine Datei. Links oben ist der Einstiegspunkt für das Dateisystem auf dem Datenträger dargestellt, der Wurzelblock (engl.: *root block*). Er zeigt auf das oberste Verzeichnis. Vom Wurzelblock aus erreicht man jedes Objekt über einen absoluten Pfad. Zum Beispiel:

/	das oberste Verzeichnis, Einträge flach, links, rechts
/flach	die gelb gefärbte Datei rechts oben
/rechts	das rechte Verzeichnis, Einträge klein, gross
/rechts/gross	die blau gefärbte Datei rechts unten
/links/unter/nix	die leere Datei links unten in der rechten Hälfte
/links/unter/leer	das leere Verzeichnis links unten

Die Systemfunktion für Dateisystemzugriffe interpretiert Pfade komponentenweise von links nach rechts. Jede Komponente wird im bisher erreichten Verzeichnis nachgeschlagen. Das Trennzeichen ganz am Anfang kennzeichnet einen absoluten Pfad, dessen erste

Komponente sich auf das obersten Verzeichnis bezieht. Ansonsten handelt es sich um einen relativen Pfad, dessen erste Komponente sich auf das Arbeitsverzeichnis der jeweiligen Instanz bezieht. Zwei Sonderfälle sind die Komponenten `.` für das bisher erreichte Verzeichnis und `..` für das übergeordnete Verzeichnis. Mit dem Kommando `cd` oder dem entsprechenden Systemaufruf kann man das Arbeitsverzeichnis ändern, auf das sich relative Pfade beziehen. Beispiele für relative Pfade, nach `cd /links/unter`:

.	das aktuelle Verzeichnis, Einträge was, nix, leer
nix	die leere Datei links unten in der rechten Hälfte
..	das Verzeichnis links mittig, einziger Eintrag unter
../..	das oberste Verzeichnis
../../rechts/klein	die grün gefärbte Datei in der zweiten Reihe

Findet die Systemfunktion eine Pfadkomponente nicht im jeweiligen Verzeichnis, kann sie den Pfad nicht auflösen und meldet einen Fehler. Ausnahme ist die letzte Komponente beim Anlegen einer neuen Datei oder eines Verzeichnisses. In diesen Fällen erzeugt die Systemfunktion den fehlenden letzten Verzeichniseintrag für das neue Objekt.

5.2. Links

Ein *Link* (dt.: Verweis) ist ein explizit erstellter Verzeichniseintrag. Das bedeutet, er entsteht nicht implizit durch Erzeugen einer Datei oder eines Verzeichnisses. Verschiedene Dateisysteme unterstützen unterschiedliche Arten von Links:

- POSIX kennt zwei Arten von Links, harte und symbolische. Das Kommando zum Erzeugen heißt `ln`. Erklärung der beiden Arten folgt gleich.
- Windows (NTFS) kennt harte Links und mehrere Varianten von symbolischen Links bzw. Junctions. Das Kommando zum Erzeugen heißt `mklink.exe`. Details dazu bitte selbst recherchieren. Der Oberbegriff lautet *Reparse Points*.

Abbildung 5.2 zeigt ein Dateisystem mit harten und symbolischen Links. Die Darstellung ist in drei Bereiche aufgeteilt. Oben links liegen die Verzeichnisse, oben rechts Dateien, beides aus Abbildung 5.1 bekannt. Symbolische Links sind ein eigener Typ von Dateisystemobjekten, hier im unteren Bereich dargestellt. Die Größe der Bereiche in der Abbildung hat nichts damit zu tun, wieviel Platz sie auf einem Datenträger einnehmen. Symbolische Links sind kleine Objekte, sie speichern nur einen Pfad.

5.2.1. Harte Links

Ein harter Link (engl.: *hard link*) ist ein normaler Verzeichniseintrag. Das Besondere an harten Links ist, dass sie für Objekte erzeugt werden, für die an anderer Stelle im Verzeichnisbaum schon ein Eintrag existiert. Nach dem Anlegen gibt es also mindestens zwei Einträge für das Objekt. Diese sind absolut gleichwertig, man kann nicht zwischen Original und Kopien unterscheiden.

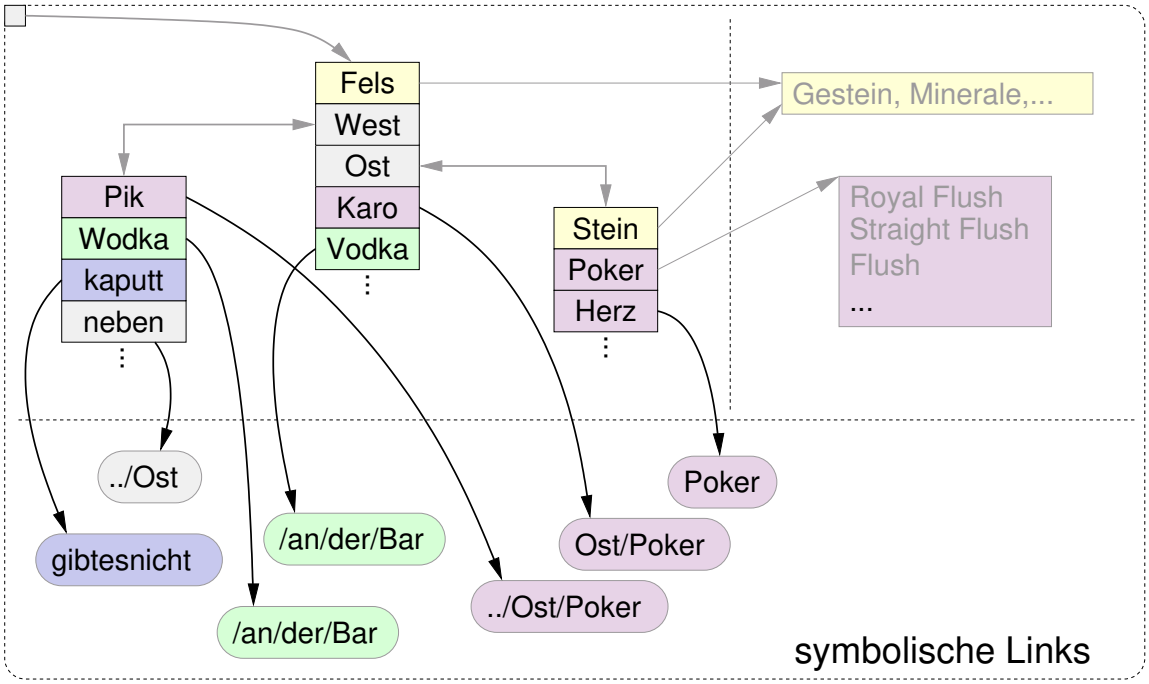


Abbildung 5.2.: Symbolische und harte Links

- Harte Links auf Dateien sind der Normalfall. Abbildung 5.2 zeigt mit `Fels` und `Stein` zwei Verzeichniseinträge, die auf die gleiche Datei verweisen. Mindestens einer davon wurde als harter Link angelegt.

- Harte Links auf Verzeichnisse kann man *nicht* mehr anlegen.[5]Das vermeidet Zyklen in der Verzeichnisstruktur, mit denen sich niemand herumschlagen möchte.[6] Beim Anlegen von Snapshots und ähnlichen Spezialoperationen können noch harte Links auf Verzeichnisse entstehen. Dabei verhindert die Spezialoperation Zyklen.

- Harte Links auf symbolische Links kann man anlegen. Um die Verwirrung in Grenzen zu halten, gehe ich nicht weiter darauf ein.

Harte Links funktionieren nur innerhalb eines Dateisystems. Sie können nicht auf oder in andere Dateisysteme zeigen. Beim Löschen von Verzeichniseinträgen bleiben Objekte so lange erhalten, bis kein Eintrag mehr darauf zeigt. Deshalb können harte Links nicht brechen, also ins Leere zeigen oder ungültig werden.
Ein Anwendungsbeispiel für harte Links ist der Paketmanager `conda`. Er verwaltet vor allem für Python–Projekte verschiedene Umgebungen im Dateisystem, die individuell ausgewählte Pakete enthalten. `conda` nutzt harte Links, um bei Bedarf das gleiche Paket in mehreren Umgebungen bereitzustellen. Nur schreibbare oder umgebungsspezifische Dateien werden kopiert und verändert.

[5]In alten Unix–Dateisystemen mussten Administratoren harte Links auf `.` und `..` anlegen, weil es keinen Systemaufruf `mkdir()` gab. HFS+ von Apple erlaubte noch das Anlegen harter Links auf Verzeichnisse, der Nachfolger APFS nicht mehr.
[6]`https://xkcd.com/981/` *Porn Folder*

5.2.2. Symbolische Links

Ein symbolischer Link oder Symlink ist ein Objekt, das einen Pfad enthält. Trifft die Systemfunktion beim Auflösen von Pfaden auf einen symbolischen Link, so löst sie zunächst diesen auf, bevor sie mit der nächsten Komponente des ursprünglichen Pfads weitermacht. Bezugspunkt für Links mit relativen Pfaden ist das Verzeichnis, in dem sich der Link befindet. Abbildung 5.2 zeigt Beispiele für verschiedene Fälle:

- Relativer Pfad zu einer Datei. Streng genommen zu einem Verzeichniseintrag für eine Datei. `Pik`, `Karo` und `Herz` sind symbolische Links auf `Poker`. Da sie in verschiedenen Bereichen des Verzeichnisbaums liegen, enthalten sie unterschiedliche relativen Pfade zum gleichen Ziel.
- Relativer Pfad zu einem Verzeichnis. Streng genommen zu einem Verzeichniseintrag für ein Verzeichnis. `neben` ist ein symbolischer Link auf `Ost`.
- Absoluter Pfad. `Wodka` und `Vodka` sind symbolische Links auf das gleiche Ziel. Bei absoluten Pfaden spielt es keine Rolle, in welchem Bereich des Verzeichnisbaums der Link liegt.
- Gebrochener Link. `kaputt` enthält einen Pfad, der sich nicht auflösen lässt.

Der Pfad eines symbolischen Links wird jeweils beim Zugriff aufgelöst. Durch Umbenennen, Löschen oder Erstellen von Dateien und Verzeichnissen kann sich das Ziel zwischen zwei Zugriffen ändern. Lässt sich der Link gerade nicht auflösen, ist er gebrochen (engl.: *dangling*). Man kann symbolische Links schon gebrochen erstellen, bis zum Zugriff mag sich die Situation schon geändert haben. Im Gegensatz dazu zeigt ein harter Link ab dem Erstellen bis zum Löschen auf das gleiche Objekt, welches über den gesamten Zeitraum hinweg existiert.
Ein Anwendungsbeispiel für symbolische Links ist die Versionierung der dynamischen Bibliotheken aus Abschnitt 3.3. In ausführbaren Dateien werden Bibliotheken mit ihrer Hauptversion referenziert, im Dateisystem liegen viele aber mit der vollständigen Versionsnummer. Ein symbolischer Link stellt den Bezug her. Die folgende Ausgabe ist leicht gekürzt. Das `l` am Anfang einer Zeile markiert einen symbolischen Link, hinter `->` steht dessen Ziel. Die Zahl in der zweiten Spalte ist die Größe des Objekts in Byte.

```
$ cd /usr/lib/x86_64-linux-gnu/
$ ls -l libacl.so.*
lrwxrwxrwx     18  libacl.so.1 -> libacl.so.1.1.2301
-rw-r--r--  34888  libacl.so.1.1.2301
```

5.3. Mounts

Sind an einem Rechner zum Beispiel eine SSD, ein USB–Stick und eine SD–Karte angeschlossen, müssen Anwendungen auf die Dateisysteme aller Speichermedien zugreifen können. Die Systemfunktion muss also Pfadnamen so auflösen, dass alle diese Dateisysteme darüber erreichbar sind. Dafür gibt es zwei Ansätze:

1. Die Dateisysteme aller Speichermedien stehen gleichberechtigt nebeneinander und tragen Namen. Die erste Komponente eines absoluten Pfads bestimmt das Dateisystem, dann folgt der Pfad innerhalb des Dateisystems. Die Namen der Dateisysteme sind Laufwerksbuchstaben (Windows, `C:`) oder Gerätenamen (AmigaOS, `hd0:`). Das erste Trennzeichen ist der Doppelpunkt, anders als das für Pfadkomponenten.

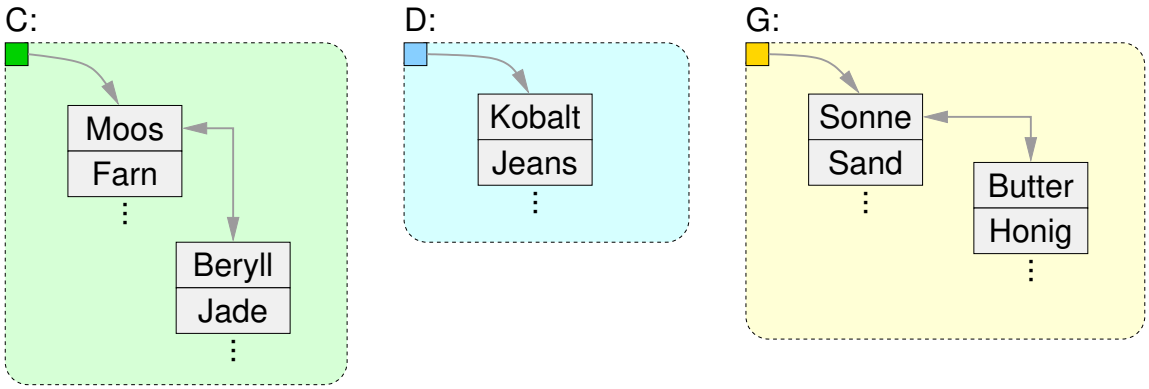


Abbildung 5.3.: Wald aus Verzeichnisbäumen

2. Eines der Dateisysteme steht als Root–Dateisystem an der Spitze. Andere Dateisysteme werden anstelle eines Verzeichnisses eingehängt. Im Englischen heißt die Operation *mount*, die Stelle an der man einhängt *mount point*. Pfade, in denen ein Mount Point vorkommt, führen von dort in das eingehängte Dateisystem.

Linux und Unixe hängen alle Dateisysteme ein, so dass die kombinierten Verzeichnisstrukturen einen Baum bilden. In Linux bezeichnet man diesen als virtuelles Dateisystem (engl.: *virtual file system, VFS*). Windows unterstützt sowohl Laufwerksbuchstaben als auch das Einhängen. Durch die Unterscheidung nach Laufwerksbuchstaben oder Gerätenamen bilden die Verzeichnisbäume einen Wald, wie in Abbildung 5.3 dargestellt. Gültige absolute Pfade in diesem Beispiel sind:

C:\
C:\Moos
D:\Jeans
G:\Sonne\Honig

Windows unterstützt auch relative Pfade mit Laufwerksbuchstaben. Für jedes Laufwerk ist ein eigenes aktuelles Verzeichnis definiert, ab dem der relative Pfad aufgelöst wird. Steht zum Beispiel das aktuelle Verzeichnis für G: auf Sonne, so sind G:Butter und G:..\Sand gültige Pfade.

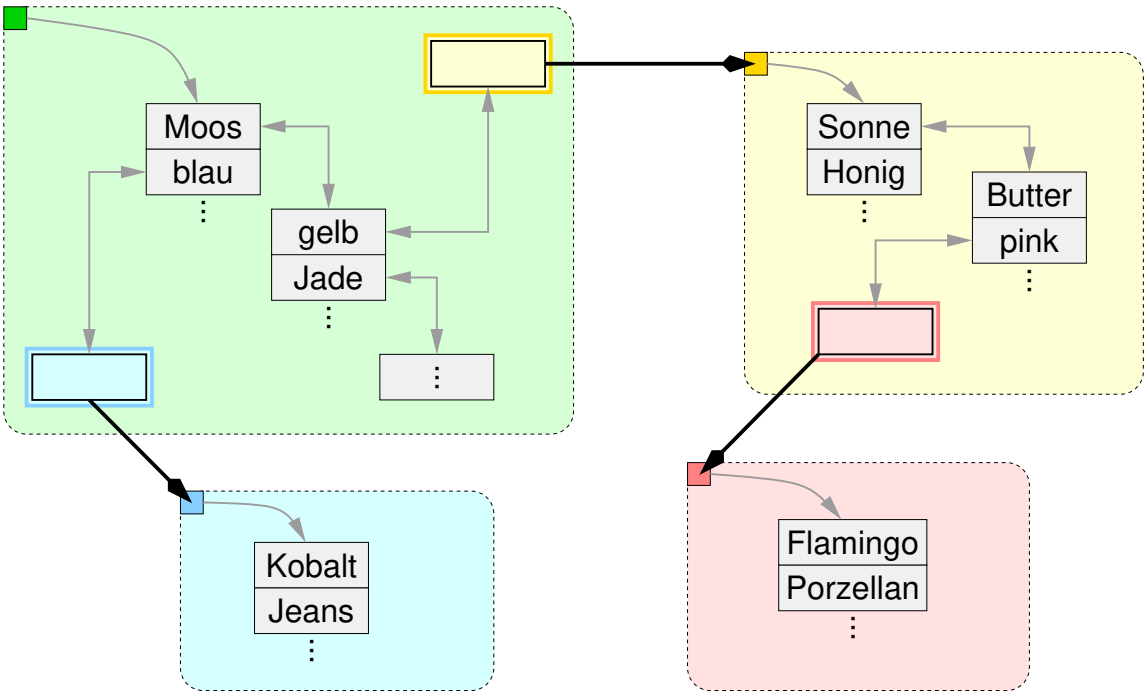


Abbildung 5.4.: Eingehängte Dateisysteme

Die Alternative mit eingehängten Dateisystemen zeigt Abbildung 5.4 Das grüne links oben ist das Root–Dateisystem. Zwei der Verzeichnisse, blau und gelb, dienen als Mount Points. An diesen Stellen überlagern das blaue Dateisystem unten bzw. das gelbe rechts die eigentlich im grünen vorhandenen Verzeichnisse. Deren ursprünglicher Inhalt ist nicht zugreifbar, solange dort andere Dateisysteme eingehängt sind. Deshalb bevorzugt man leere Verzeichnisse als Mount Points. In der Abbildung ist außerdem noch das pinke Dateisystem rechts unten an einem Mount Point im gelben Dateisystem eingehängt. Gültige absolute Pfade in diesem Beispiel sind:

```
/
/Moos
/blau/
/Moos/gelb/Honig
/Moos/gelb/Sonne/pink/Flamingo
```

Die Verwaltung eingehängter Dateisysteme liegt bei der gleichen Systemfunktion wie das Auflösen von Pfaden. Das Einhängen eines Dateisystems verändert weder Inhalte noch Metadaten im Dateisystem des Mount Points. Aber die Systemfunktion prüft beim Auflösen von Pfaden für jede Komponente, ob es sich um einen Mount Point handelt. Erst danach greift sie auf die Verzeichniseinträge des passenden Dateisystems zu. Das funktioniert auch, wenn `..` in das Elterverzeichnis eines Mount Points führt.

5.4. Operationen

Pfade beruhen in erster Linie auf Verzeichniseinträgen. Typische Operationen für Verzeichniseinträge sind:

Erzeugen: Beim Anlegen einer Datei, eines Verzeichnisses, eines symbolischen oder harten Links entsteht ein neuer Eintrag. Außer bei den harten Links muss zunächst das Objekt erstellt werden, auf das der neue Eintrag verweist.

Löschen: Das Löschen eines Verzeichniseintrags führt automatisch zum Löschen des referenzierten Objekts, sofern kein anderer Eintrag darauf verweist. Dateien bleiben erhalten, solange noch ein harter Link, also ein Verzeichniseintrag, auf sie verweist. Verzeichnisse können nur gelöscht werden, wenn sie keine Einträge mehr enthalten. Verzeichnisbäume zu löschen ist also eine rekursive Operation.

Umbenennen: Den Namen eines Verzeichniseintrags zu ändern ist eine relativ einfache und schnelle Operation, da keine großen Datenmengen bewegt werden müssen. Namen innerhalb eines Verzeichnisses müssen eindeutig sein, also ist ein Vergleich mit den anderen Namen notwendig.

Verschieben: Ein Objekt im gleichen Dateisystem zu verschieben erfordert das Anlegen eines neuen Verzeichniseintrags und das Löschen des alten. Das führt nur zu wenig Mehraufwand gegenüber dem Umbenennen.
Zum Verschieben in ein anderes Dateisystem muss das referenzierte Objekt ins Zielsystem kopiert und anschließend im Quellsystem gelöscht werden. Bei großen Dateien oder Verzeichnisbäumen führt das zu erheblichem Aufwand. Zeigt noch ein harter Link auf die Quelldatei, bleibt diese im Quellsystem erhalten. Der Umgang mit symbolischen und harten Links beim Umkopieren von Verzeichnisbäumen ist situationsabhängig. Man sollte damit rechnen, dass mehrere harte Links auf die gleiche Datei im Quellsystem zu mehreren Kopien der Datei im Zielsystem führen.

Auch das Ein- und Aushängen von Dateisystemen wirkt sich auf die gültigen Pfade aus. Beim Aushängen ist zu beachten, dass Änderungen am Dateisystem noch im Hauptspeicher gepuffert sein können. Erst wenn solche Änderungen auf das Speichermedium geschrieben sind, kann man dieses ohnen Datenverlust entfernen.

5.5. Knackpunkte

- Pfade sind Zeichenketten, die Wege durch die Verzeichnisstruktur von Dateisystemen beschreiben und zu einem bestimmten Verzeichniseintrag führen. Das Auflösen von Pfaden ist eine wichtige Aufgabe der Systemfunktion, die den Zugriff auf Dateisysteme ermöglicht.
- Die Pfadkomponente `.` steht für das aktuelle Verzeichnis im Verlauf der Pfadauflösung, `..` für das Elterverzeichnis. Verzeichnisse bilden einen Baum, das Elterverzeichnis ist also eindeutig.
- In jedem Verzeichnis sind die Namen aller Einträge eindeutig.
- Verzeichniseinträge verweisen auf eine Datei, ein Verzeichnis, einen symbolischen Link oder ein anderes Objekte im Dateisystem, zum Beispiel einen *Reparse Point*. In der Vorlesung betrachte ich nur die ersten drei Fällen.
- Ein symbolischer Link oder *Symlink* enthält einen Pfad, der meist mit aufgelöst wird, wenn der Link als Komponente eines anderen Pfads auftritt.
- Ein harter Link oder *Hard Link* ist ein Verzeichniseintrag, der meist auf eine Datei verweist. Durch harte Links kann man explizit mehrere Einträge für die gleiche Datei erzeugen. Das spart Platz gegenüber Kopien.
- Symbolische Links können brechen, also einen gerade nicht auflösbaren und deshalb ungültigen Pfad enthalten. Harte Links verweisen direkt auf eine Datei, nicht auf einen anderen Verzeichniseintrag. Sie können deshalb nicht brechen.
- Das Betriebssystem muss den Zugriff auf verschiedene, gleichzeitig angeschlossene Dateisysteme erlauben. Die gängigen Verfahren dazu sind Laufwerksbuchstaben und Mount Points.
- Mit dem Löschen eines Verzeichniseintrags verschwindet auch das Objekt, auf das er verweist. Außer es gibt noch einen anderen Verzeichniseintrag, welcher auf das gleiche Objekt verweist. Also einen harten Link auf die gleiche Datei.
- Verzeichnisse kann man nur löschen, wenn sie leer sind.
- Verschieben innerhalb eines Dateisystems ist eine relativ einfache Operation, weil sie nur einen neuen Verzeichniseintrag erzeugt und den alten löscht. Verschieben in ein anderes Dateisystem bedeutet ggfs. rekursives Kopieren und viel Aufwand.

6. Programmierschnittstellen

Programmierschnittstellen oder APIs (engl.: *Application Programming Interfaces*) liegen an den Grenzen gekapselter Software. In Betriebssystemen existieren scharfe Grenzen zwischen verschiedenen Instanzen und der Infrastruktur. Damit Instanzen die Infrastuktur nutzen und miteinander arbeiten können, brauchen sie entsprechende APIs. Im Folgenden stelle ich drei Arten von APIs vor. Abschnitt 6.1 betrachtet Bibliotheken, die im Adressraum der aufrufenden Instanz arbeiten. Abschnitt 6.2 beschreibt die Schnittstelle zur Infrastuktur. Abschnitt 6.3 geht auf Protokolle zwischen Instanzen ein. Abschnitt 6.4 fasst die wichtigsten Punkte dieses Kapitels zusammen. Die genannten drei Arten von APIs entsprechen den drei Bereichen, in denen Systemfunktionen einen Teil ihrer Aufgaben erledigen können:

- Im Adressraum der aufrufenden Instanz.
- Im Kern des Betriebssystems.
- In dedizierten Instanzen für die Systemfunktion.

Um den sperrigen Begriff Programmierschnittstelle zu vermeiden, schreibe ich fortan entweder Schnittstelle oder API. Es gibt auch völlig andere Arten von Schnittstellen, zum Beispiel Benutzerschnittstellen oder Peripherieschnittstellen. Diese sind hier offensichtlich nicht gemeint. Als Artikel für API verwende ich entweder „die“ wie für Schnittstelle oder „das“ wie für Interface.[1]

6.1. Bibliotheken

Bibliotheken als Software–Module sind bereits aus Kapitel 3 bekannt. Sie werden in den Adressraum einer Instanz geladen, entweder als Teil des Programms bei statisch gebundenen Bibliotheken, oder in eigene Segmente bei dynamisch gebundenen Bibliotheken. Abbildung 6.1 zeigt zur Veranschaulichung eine Instanz mit einem Programm und drei dynamisch gebundenen Bibliotheken. Blaue Pfeile stehen für Aufrufe zwischen diesen Modulen. Die technische Aufteilung in mehrere Segmente pro Modul ist nicht eingezeichnet, weil sie hier keine Rolle spielt.
Die API einer Bibliothek wird mit den Mitteln einer Programmiersprache definiert: Funktionen, Klassen, Methoden und so weiter. Der Compiler für die Programmiersprache übersetzt Aufrufe in Binärcode. Eine Aufrufkonvention legt fest, auf welche Weise der generierte Code auf einer bestimmten Prozessorarchitektur Parameter und Ergebnisse übergibt, oder wie Exceptions geworfen und gefangen werden. Diese Aufrufkonvention macht aus der Programmierschnittstelle, der API, eine prozessorspezifische Binärschnittstelle, ein ABI (engl.: *Application Binary Interface*). Anwendungsentwickler müssen sich mit den

[1] API spreche ich mal als ein Wort (Ahpie), mal englisch buchstabiert (Äi-Pi-Ei). Es lebe die Vielfalt!

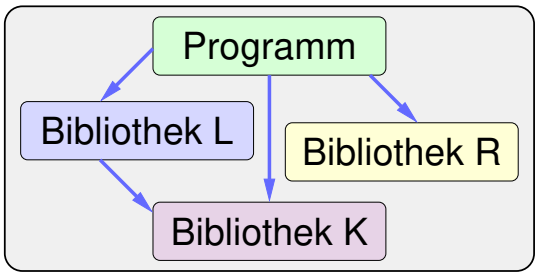


Abbildung 6.1.: Instanz mit Bibliotheken

Details des ABI aber nicht auseinandersetzen, sie nutzen einfach die API in der jeweiligen Programmiersprache.
Zur Laufzeit erfolgen Aufrufe in Bibliotheken durch Sprungbefehle im Befehlsstrom. Da aller Code im gleichen Adressraum liegt, dürfen verschiedene geladene Bibliotheken einander aufrufen, oder über Funktionszeiger auch in den Code des Programms zurückspringen (engl.: *callback*). Bibliotheken haben Zugriff auf alle Daten im Adressraum. So können sie die Halde verwalten, die Pufferung von Ein-/Ausgabeströmen übernehmen, und Mathe-, Krypto-, oder Kompressionsalgorithmen auf die Daten der Instanz anwenden.

6.2. Kern

Die Schnittstelle des Kerns liegt an der Grenze zu den Instanzen, wie in Abbildung 6.2 dargestellt. Instanzen können den Kern aufrufen und bestimmte Aktionen anfordern. Der Kern entscheidet, nach Prüfung von Parametern, Zugriffsrechten, verfügbaren Resourcen und anderen Kriterien, ob und wie er diese Forderungen erfüllt. Das ist ein wichtiger Unterschied zu Bibliotheken. Dort können Aufrufer an beliebige Stellen in den Code springen und auch beliebige Manipulationen an den Daten durchführen. Bibliotheken arbeiten ungeschützt im Adressraum des Aufrufers. Der Kern dagegen schützt sich vor ungültigen Zugriffen aus Instanzen heraus.
Instanzen können in unterschiedlichen Programmiersprachen geschrieben sein. Auch im Kern kommt neben Maschinensprache mindestens eine systemnahe, übersetzte Programmiersprache zum Einsatz. Zur Entkopplung verwendet der Kern eine Binärschnittstelle, ein ABI. Dabei ist es sinnvoll, für die Übergabe von Parametern und Ergebnissen auf die Aufrufkonventionen einer Programmiersprache wie C oder Rust zurückzugreifen. Exceptions gibt es nur in manchen Programmiersprachen, sie gehören nicht in eine Kernschnittstelle. Informationen über Erfolg, Misserfolg und aufgetretene Fehler gibt der Kern als Ergebnisse zurück, häufig in Form eines Statuscodes.
Instanzen können nicht in den Kern springen, wie sie es bei einer Bibliothek tun. Statt dessen bereiten sie die Parameter eines Aufrufs vor und lösen dann eine Unterbrechung aus, einen Software–Interrupt. Diese Unterbrechung veranlasst den Kern, den Aufruf zu verarbeiten. Sobald der unterbrochene Befehlsstrom der Instanz weiterläuft, kann er die Ergebnisse des Kernaufrufs verwenden. Die Rollen von Befehlsströmen und Unterbrechungen erklärt Kapitel 12, Details zu Kernaufrufen liefert Kapitel 16.

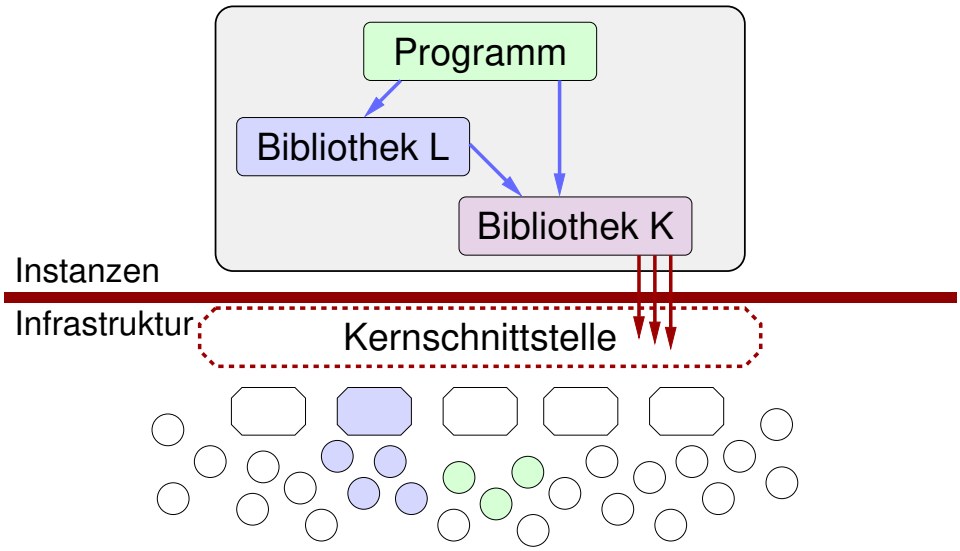


Abbildung 6.2.: Kernschnittstelle

Der Aufruf des Kern–ABI erfordert Assemblercode in der Instanz. Diesen kapselt man in einer Bibliothek, welche als Adapter zwischen den Aufrufkonventionen der Programmiersprache und des Kerns fungiert. Hier könnte man zum Beispiel auch Fehlersituationen erkennen und dafür Exceptions auslösen, falls die Programmiersprache das hergibt. In Abbildung 6.2 kapselt Bibliothek K den Assemblercode. Andere Bibliotheken oder das Programm nutzen Bibliothek K, um Kernaufrufe durchzuführen.
Im Infrastrukturbereich von Abbildung 6.2 stehen die Achtecke für verschiedene Systemfunktionen, die im Kern angesiedelt sind. Die Kreise repräsentieren Kernobjekte, also nach außen sichtbare Resourcen im Kern. Eine Systemfunktion könnte zum Beispiel die Kommunikation über Kanäle anbieten, wie sie bei den Interaktionsdiagrammen in Kapitel 2 vorkommen. Dann ist jeder einzelne Kanal ein Kernobjekt, welches Operationen zum Senden und Empfangen von Nachrichten unterstützt. Um eine solche Systemfunktion für Instanzen bequem nutzbar zu machen, kann man wiederum eine Bibliothek bereitstellen. In der Abbildung ist das Bibliothek L. Für die Kernaufrufe greift sie auf Bibliothek K zurück. Andere Systemfunktionen ruft das Programm vielleicht direkt über Bibliothek K auf.

Anmerkung: *Ich beschreibe Systemfunktionen und die Kernschnittstelle gerne aus einer objektorientierten Perspektive: Kernobjekte, Operationen (Methoden) und Attribute. Das ist keine technische Einschränkung, sondern nur eine Interpretation. Der Aufruf einer Methode an einem Objekt ist gleichwertig mit dem Aufruf einer Funktion oder Prozedur, die als erstes Argument die Kennung einer Datenstruktur (`this`, `self`) erwartet. Bei der Kernschnittstelle ist dabei wichtig, dass keine Adresse sondern eine Kennung (ID) für das Kernobjekts verwendet wird. Instanzen sollten keine Adressen aus der Infrastruktur kennen. Das verbessert die Trennung der Bereiche und erschwert Attacken.*

6.3. Protokolle

Wenn Instanzen miteinander arbeiten, nutzen sie dazu direkt oder indirekt Interaktionsmechanismen der Infrastruktur. Der englische Oberbegriff für solche Mechanismen lautet IPC, *Inter–Process Communication*.[2] Alle beteiligten Instanzen halten sich an ein gemeinsames Protokoll. Dieses regelt unter anderem:

- Welcher Kommunikationsmechanismus wird verwendet?
- Wie finden die Instanzen einander?
- Welche Instanz kommuniziert in welcher Situation welche Informationen?

Ein häufig anzutreffendes Schema sind Dienste, die ihre Funktionen als Server anbieten und Anwendungen, die diese als Clients in Anspruch nehmen. Zum Beispiel beschreibt das *Language Server Protocol* (LSP),[3] wie Editoren oder Entwicklungsumgebungen einen auf dem gleichen Rechner laufenden Language Server nutzen, um Komfortfunktionen für die vom Server unterstützte Programmiersprache anzubieten.
Meist sind Protokolle geschichtet. So setzt LSP auf JSON-RPC auf, welches seinerseits über unterschiedliche Transportmechanismen wie Pipes oder Sockets funktioniert. Diese Transportmechanismen liefert die Infrastruktur. Protokolle verbinden auch Instanzen auf verschiedenen Rechnern über Netzwerke. Zum Beispiel kann ein Server eine REST–API anbieten, welche auf HTTPS aufsetzt. Von der Infrastruktur braucht es dazu TCP/IP oder UDP/IP als Transportmechanismus, je nach HTTP–Version.

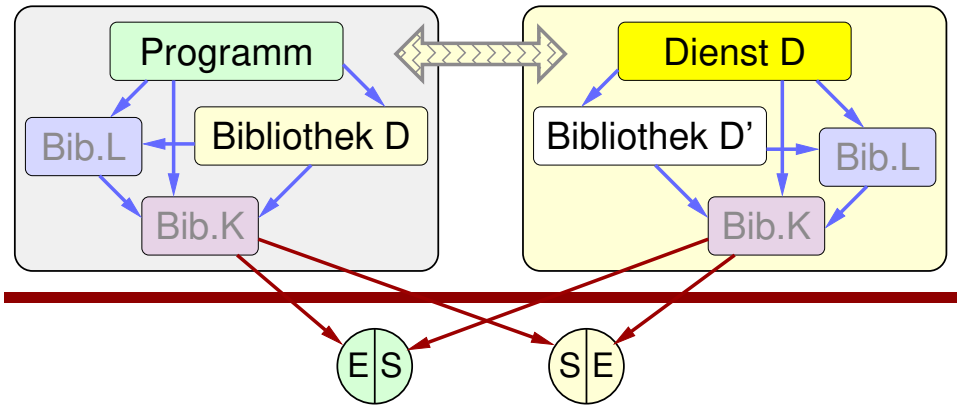


Abbildung 6.3.: Instanz zu Instanz über Bibliotheken

Abbildung 6.3 greift das Interaktionsmuster Client/Server aus Abschnitt 2.2 als Beispiel auf. Die linke Instanz ist ein Client, die rechte der Server, der Doppelpfeil dazwischen steht für das Protokoll. Der Server bietet einen Dienst D an, seine Schnittstelle ist im Protokoll festgelegt. Um den Dienst bequem nutzbar zu machen gibt es eine Bibliothek D, welche die Client–Seite des Protokolls kapselt. Das Gegenstück dazu ist die Bibliothek D' rechts, mit der Server–Seite des Protokolls. Als Transportmechanismus bietet die Infrastruktur

[2] dabei steht *process* für Instanzen oder Adressräume, nicht für Befehlsströme
[3] `https://microsoft.github.io/language-server-protocol/`, abgerufen 2024–06–22

hier Kanäle an. Die Bibliotheken L und K sind schon aus den vorhergehenden Abschnitten bekannt. Sie kommen auf beiden Seiten zum Einsatz, um den Transportmechanismus zu nutzen.
Systemdienste in eigenen Instanzen (engl.: *daemons*) eignen sich besonders für Funktionen, die mehreren Anwendungen gemeinsam zur Verfügung stehen, die man aber nicht in die Infrastruktur stopfen will. Zum Beispiel die Zwischenablage (engl.: *clipboard*) einer graphischen Benutzerschnittstelle, die Ausgabe von Tönen aus verschiedenen Quellen oder das Ausdrucken von Dokumenten. Der Übergang zu sogenannter *Middleware* wie Datenbanken oder Message Queueing Systemen ist dabei fließend.

Bibliotheken spielen eine zentrale Rolle als Programmierschnittstellen, denn nur Bibliotheken kann man aus Anwendungscode heraus direkt aufrufen. Alle anderen Arten von Schnittstellen, zur Infrastruktur, zu anderen Instanzen, lokal, über ein Netzwerk, zu angeschlossenen Peripheriegeräten, zu Benutzern, erreicht man über Bibliotheken. Sobald es eine Bibliothek für eine Systemfunktion gibt, spielt es für Anwendungsentwicklerinnen kaum noch eine Rolle, wo genau die verschiedenen Teile der Systemfunktion implementiert sind. Bei der Geschwindigkeit und in Fehlersituationen können sich aber Unterschiede zeigen. Es schadet also nicht zu wissen, ob eine Systemfunktion von einem Systemdienst abhängt, der vielleicht gar nicht gestartet ist.
Umgekehrt hilft es bei der Strukturierung von Anwendungscode, wenn man fehlende Bibliotheken für Schnittstellen selbst erstellt. Insbesondere bei Protokollen kann man häufig auf Bibliotheken für untere Protokollebenen zurückgreifen, aber nicht für die oberste. Zum Beispiel gibt es HTTP(S)–Clients für alle gängigen Programmiersprachen, während man Bibliotheken für anwendungsspezifische REST–APIs aus der Spezifikation generieren oder von Hand implementieren muss.

6.4. Knackpunkte

- *Application Programming Interface* (API) ist ein weit gefasster Begriffe, der für verschiedene Arten von Programmierschnittstellen verwendet wird.
- Bibliotheken implementieren APIs, die man aus der jeweiligen Programmiersprache heraus direkt aufrufen kann.
- Bibliotheken arbeiten im Adressraum der aufrufenden Instanz. Aufrufe sind Sprünge im übersetzten Code, die der Prozessor direkt ausführt.
- Die Infrastruktur, der Kern, liegt außerhalb der Instanzen. Aufrufe erfolgen über Software–Interrupts. Dazu braucht es Assembler–Code in der Instanz.
- Die Schnittstelle des Kerns ist unabhängig von Programmiersprachen definiert, als *Application Binary Interface* (ABI).
- Eine Bibliothek adaptiert die Aufrufkonventionen einer bestimmten Programmiersprache an das ABI des Kerns und kapselt den Assembler–Code.
- Weitere Bibliotheken stellen bequeme APIs für bestimmte Systemfunktionen bereit.
- Protokolle regeln Aufrufe oder andere Interaktionen zwischen Instanzen.
- Protokolle setzen meist auf anderen, niedrigeren Protokollen auf.
- Protokolle brauchen mindestens einen Interaktionsmechanismus zwischen Instanzen, den die Infrastruktur bereitstellt.
- Protokolle kann man in Bibliotheken kapseln, die bequeme APIs bereitstellen.
- Im Anwendungscode ruft man letztlich ausschließlich APIs in Bibliotheken.

Abbildung 6.4 zeigt die genannten drei Arten von APIs im Vergleich. Blaue Pfeile stehen für Aufrufe innerhalb des Adressraums. Rote Pfeile stellen Aufrufe des Kerns dar. Der gelbe Doppelpfeil symbolisiert ein Protokoll.

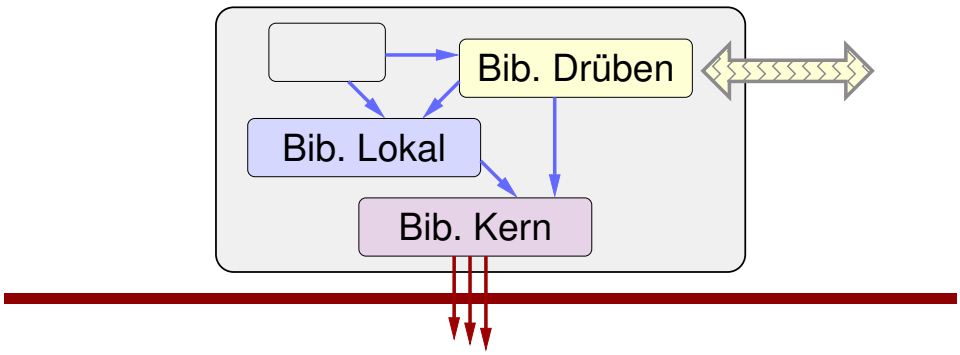


Abbildung 6.4.: Vergleich der drei Möglichkeiten

7. Die Infrastruktur

Die Begriffe *Instanzen* und *Infrastruktur* wurden bereits in Kapitel 1 eingeführt, *Kern* mit Abschnittt 6.2. Der Kern (engl.: *kernel*) und die Infrastruktur sind praktisch gleichbedeutend. Es handelt sich um den zentralen Teil eines jeden Betriebssystems, welches mehrere Instanzen gleichzeitig ausführen kann. Die Infrastruktur hat im Vergleich zu Instanzen erweiterte Zugriffsmöglichkeiten. Diese verwendet sie einerseits um Instanzen abzuschotten, andererseits um Zusammenarbeit von Instanzen zu ermöglichen.
Der folgende Abschnitt 7.1 umreißt die Aufgaben der Infrastruktur. Abschnitt 7.2 beantwortet die Frage: Was ist der Kern? Abschnitt 7.3 stellt Kernobjekte als eine Möglichkeit vor, die Schnittstelle der Infrastruktur gegenüber den Instanzen zu strukturieren. Abschnitt 7.4 fasst die wichtigsten Punkte dieses Kapitels zusammen.

7.1. Aufgaben

Eine Instanz, also ein laufendes Programm, braucht auf jeden Fall:

Speicher für Code und Daten. Code besteht aus Befehlen, also Anweisungen für einen Prozessor. Zu den Daten gehören erhaltene Eingabe- und berechnete Ausgabedaten, interne Daten während des Programmlaufs und mehr.

Rechenzeit um Befehle auszuführen. Jede Instanz definiert mindestens einen, manche auch mehrere Befehlsströme, die im Rechner ablaufen sollen.

Kontakt zum Umfeld. Dieses liefert Eingaben und nimmt Ausgaben entgegen. Zum Umfeld zählen andere Instanzen, Peripheriegeräte für Benutzer, Speichermedien und vieles mehr. Ohne Kontakt zum Umfeld wäre die Ausführung eines Programms sinnlos.

Abbildung 7.1.: Aufgaben der Infrastruktur

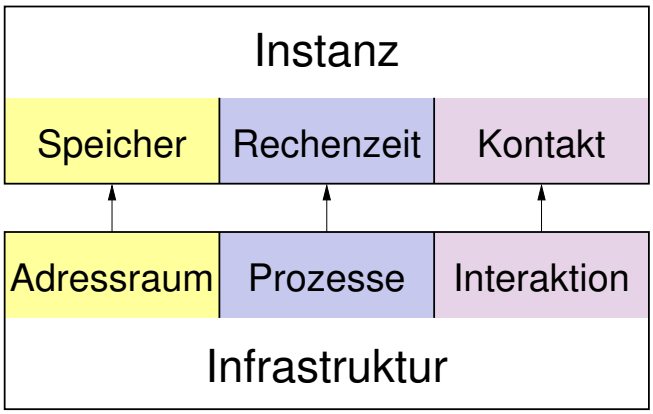


Abbildung 7.1 führt die Mittel auf, mit denen die Infrastruktur diesen Bedürfnissen Rechnung trägt:

Adressräume enthalten Speicher

Prozesse bekommen Rechenzeit und führen damit Befehlsströme aus

Interaktion vermittelt den Kontakt zum Umfeld

Jede Instanz existiert in einem eigenen Adressraum, in dem Segmente mit ihren Speicherinhalten liegen. Das ist bereits aus Kapitel 3 bekannt. Die Prozesse einer Instanz treiben Befehlsströme in deren Adressraum voran. Das folgende Kapitel 8 erläutert den Umgang des Kerns mit Prozessen. Kontakt mit ihrem Umfeld erhalten Instanzen über Interaktionsmechanismen des Kerns. Dazu zählen die Ströme aus Kapitel 4. Einige andere, systemnahe Möglichkeiten zur Interaktion stelle ich ab Kapitel 9 vor.

7.2. Was ist der Kern?

Der Kern eines Betriebssystems ist Software und überwiegend in einer systemnahen Hochsprache wie C geschrieben. Das heißt der Quelltext wird von einem Compiler übersetzt, ebenso wie bei Programmen oder Bibliotheken. Zur Laufzeit hat der Kern also ähnliche Bedürfnisse wie eine Instanz: Speicher, Rechenzeit, Kontakt zum Umfeld. Aufgrund der privilegierten Rolle der Infrastruktur gibt es aber auch grundlegende Unterschiede.

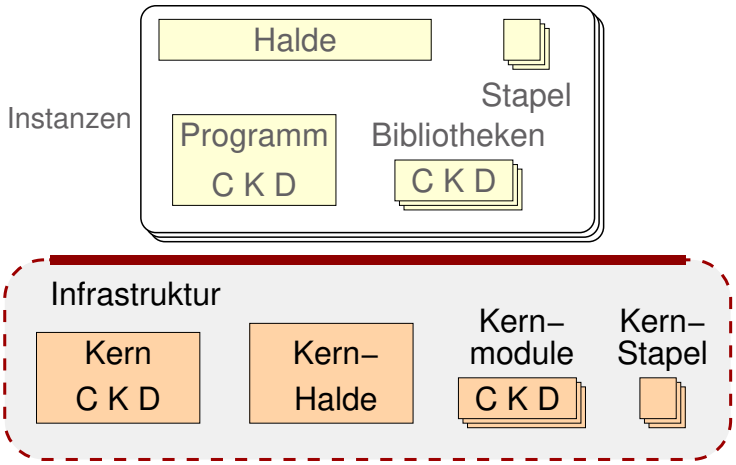


Abbildung 7.2.: Beschaffenheit der Infrastruktur

Abbildung 7.2 stellt eine Instanz und die Infrastruktur gegenüber. In der Instanz liegt ein Programm, in der Infrastruktur der Kern. Der Kern braucht eine Halde für langlebige Kernobjekte sowie mehrere Stapel für parallel laufende Befehlsströme. Bibliotheken für Instanzen kann der Kern im Allgemeinen nicht verwenden, weil diese Kernaufrufe auslösen. Zum Beispiel könnte eine Bibliothek zur Speicherverwaltung den Kern aufrufen, um die Halde der Instanz zu vergrößern. Die Kernschnittstelle unterstützt aber nur Aufrufe aus Instanzen heraus. Das Gegenstück zu dynamisch gebundenen Bibliotheken bei Instanzen sind deshalb Kernmodule, die in der Infrastruktur funktionieren.

Ebenso wie jede Instanz ihren eigenen Adressraum hat, gibt es auch für die Infrastruktur einen speziellen Kernadressraum. Während Befehlsströme von Instanzen aber nur auf Speicher im eigenen Adressraum zugreifen dürfen, muss die Infrastruktur in bestimmten Fällen Speicherinhalte von Instanzen auslesen oder Daten dort ablegen. Deshalb ist die Umrandung des Kernadressraums in Abbildung 7.2 gestrichelt statt durchgezogen.
Der Kern braucht neben Speicher auch Rechenzeit und Kontakt zu seinem Umfeld. Auf die Rechenzeit gehen Kapitel 12 und Abschnitt 16.3 ein. Kontakt zu den Instanzen hält der Kern, indem diese ihn aufrufen. Das bespricht Kapitel 16. Die Ansteuerung der Peripherie, also Kontakt zur Außenwelt, ist derzeit kein Thema meiner Vorlesung.

Begriffsklärung: *Infrastruktur und Kern sind weitgehend, aber nicht exakt gleichbedeutend. Die Infrastruktur ist das Gegenstück zu den Instanzen, ein abstraktes Konzept. Mit Kern meine ich die Implementierung der Infrastruktur. Meist spielt dieser kleine Unterschied keine Rolle. Wenn eine Instanz die Infrastruktur aufruft, ruft sie auch den Kern. Bei zusammengesetzten Begriffen gewinnt die Lesbarkeit: Kernschnittstelle, Kernobjekt,...*

7.3. Kernobjekte

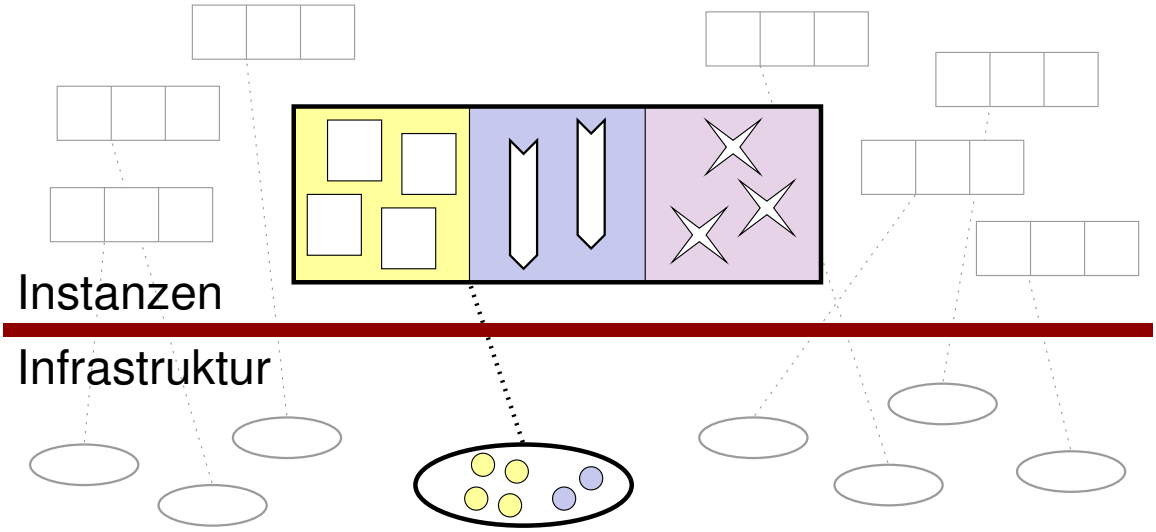


Abbildung 7.3.: Kernobjekte zur Verwaltung

Abbildung 7.3 zeigt mehrere Instanzen. Anders als in Kapitel 1 sind sie hier als dreigeteilte Rechtecke gezeichnet, was für die drei Bedürfnisse steht: Speicher, Rechenzeit und Kontakt zum Umfeld. Eine der Instanzen ist weiter aufgeschlüsselt. Die Symbole darin repräsentieren beispielhaft vier Speicherbereiche, zwei Prozesse und drei Kontaktereignisse. Der Kern kennt alle Instanzen und ihre Bedürfnisse in Form von Kernobjekten. Für jeden Adressraum, jeden Speicherbereich darin, jeden Prozess erzeugt er ein Objekt. Das sind die Ovale und Kreise im unteren Teil der Abbildung. Mit Hilfe dieser Objekte verwaltet der Kern Speicher und Rechenzeit.

Kernobjekte sind Datenstrukturen in der Halde des Kerns. Instanzen haben keinen direkten Zugriff auf diese Halde. Sie erhalten keine Adressen von Kernobjekten, sondern nur neutrale Kennungen, mit denen der Kern bei Aufrufen die Objekte wiederfindet. Deshalb kann ein Kernobjekt auch aus mehreren, verzeigerten Einzelstrukturen bestehen, wenn das für die Implementierung sinnvoll ist. Interne Attribute und Methoden von Kernobjekten bleiben vor den Instanzen verborgen, indem man sie an der Kernschnittstelle einfach nicht anbietet.

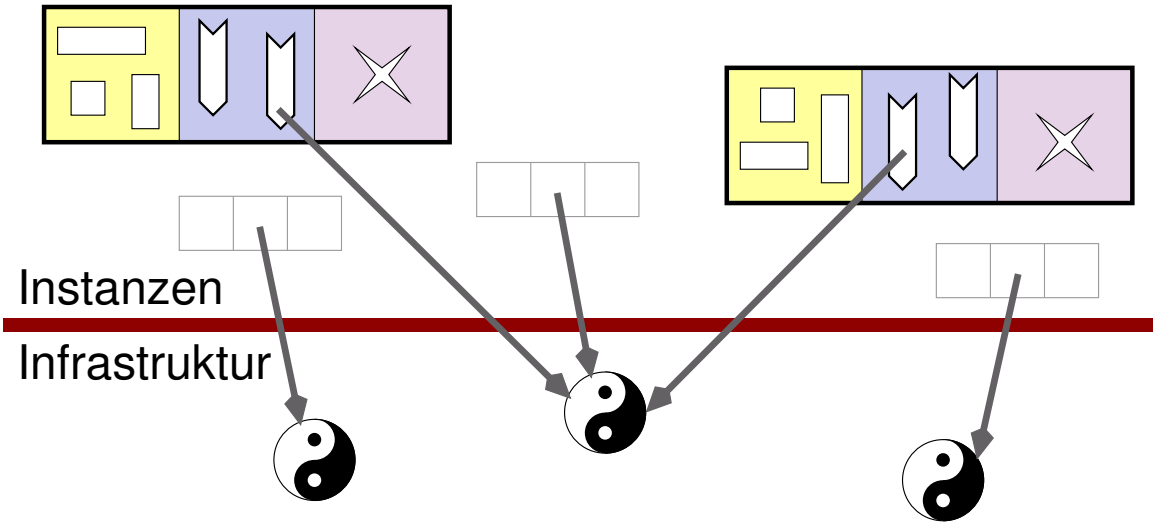


Abbildung 7.4.: Kernobjekte zur Interaktion

Kontakt zum Umfeld wird nicht verwaltet, sondern vermittelt. Auch dazu dienen Kernobjekte. In Abbildung 7.4 sind drei solche Objekte als Yin–Yang–Symbole dargestellt. In den Beispielen hier im Buch bieten Kernobjekte zur Interaktion jeweils zwei komplementäre Operationen an: Senden und Empfangen, Sperren und Entsperren oder dergleichen. In der Aufgabensammlung zur Vorlesung beschreibe ich auch andere Fälle, die aber ähnlich funktionieren.
Zur Interaktion rufen die Befehlsströme von Instanzen eine der Operationen an einem der Interaktionsobjekte auf. Erkennt der Kern zueinander passende Aufrufe, vermittelt er den Kontakt. Das führt zu einem Ereignis, auf das die beteiligten Instanzen bzw. Befehlsströme reagieren können. Am mittleren Kernobjekt in Abbildung 7.4 sind zwei Aufrufe der weißen Seite und einer der schwarzen Seite eingezeichnet. Der Kern hat hier die Aufrufe der beiden groß eingezeichneten Instanzen kombiniert. Deshalb ist dort jeweils ein Kontaktereignis als Stern zu sehen. Die anderen drei Aufrufe der klein gezeichneten Instanzen warten noch auf ein Gegenstück.
Häufig ist der Zeitpunkt des Ereignisses nur für eine Seite relevant. Der Empfänger einer Nachricht muss erfahren, dass sie eingetroffen ist, um sie zu verarbeiten. Für den Sender spielt es nicht unbedingt eine Rolle, wann genau seine Nachricht ankommt. Wer eine Sperre setzt, muss wissen, wann sie zugeteilt wird. Wer eine Sperre freigibt kümmert sich nicht darum, ob sie gleich oder erst später wieder gesetzt wird.

7.4. Knackpunkte

- „Kern“ und „Infrastruktur“ meinen praktisch das Gleiche.
- Der Kern ist die Implementierung der Infrastruktur.
- Der Kern bedient die Grundbedürfnisse von Instanzen:
  - **Speicher** in Adressräumen
  - **Rechenzeit** für Prozesse
  - **Kontakt zum Umfeld** durch Interaktion
- Der Kern ist Software: Code, Konstanten, statische Daten, Halde, Stapel
- Der Kern hat einen eigenen Adressraum, geschützt vor Instanzen.
- Der Kern kann aus seinem Adressraum herausgreifen, im Gegensatz zu Instanzen.
- Kernobjekte strukturieren die Schnittstelle des Kerns zu den Instanzen.
  - Kernobjekte zur Verwaltung von Speicher und Rechenzeit.
  - Kernobjekte zur Vermittlung von Kontakten, also zur Interaktion.
- Interaktionsobjekte bieten häufig zwei komplementäre Operationen an:
  - Senden und Empfangen
  - Sperren und Entsperren
  - …

8. Prozesse

Ein *Prozess* (engl.: *thread*) ist ein Befehlsstrom, den das Betriebssystem abarbeitet. Die Befehle darin stammen aus einem Programm. Laufen auf einem System mehrere Programme nebeneinander, dann hat jedes davon mindestens einen Befehlsstrom. Es existieren also mehrere Prozesse gleichzeitig. Das Betriebssystem nutzt die vorhandenen Prozessoren, um alle aktiven Prozesse voranzutreiben. Ein Prozessor springt dabei von einem Befehlsstrom zum nächsten. Das ist vergleichbar mit verschiedenen Handlungssträngen in einem Buch, zwischen denen der Autor hin- und herspringt.[1]

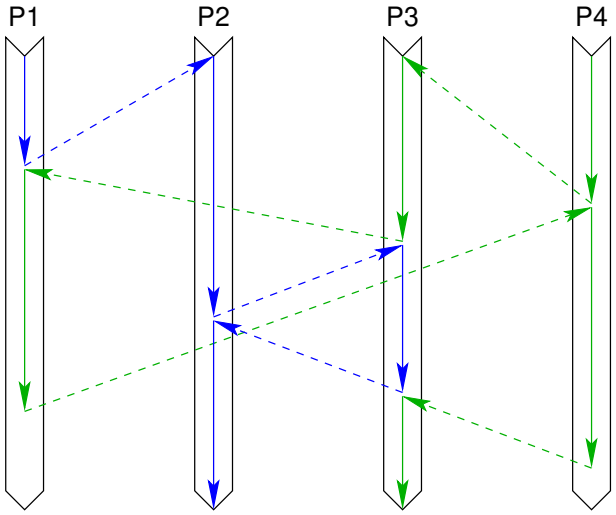


Abbildung 8.1.: Befehlsströme

Abbildung 8.1 veranschaulicht das Prinzip. Sie zeigt die Befehlsströme P1 bis P4 als breite, vertikale Streifen. Jeder Befehlsstrom verläuft von oben nach unten. Im System arbeiten zwei Prozessoren, dargestellt als blaue und grüne Pfeile. Der blaue Prozessor arbeitet zuerst an P1, springt dann zu P2, weiter zu P3 und wieder zu P2. Der grüne Prozessor beginnt mit P4, springt zu P3, dann zu P1, zurück zu P4 und noch einmal zu P3. Damit sind die abgebildeten Ausschnitte der beiden mittleren Befehlsströme abgearbeitet. Im weiteren Verlauf würden die Prozessoren wieder die äußeren Befehlsströme aufgreifen. Am Beispiel von P1 und P3 ist zu erkennen, dass ein Befehlsstrom auch von verschiedenen Prozessoren vorangetrieben wird. Allerdings darf zu jedem Zeitpunkt nur ein Prozessor daran arbeiten. Springt dieser weg, dann kann ein anderer Prozessor den Befehlsstrom an exakt der gleichen Stelle übernehmen.

[1] zum Beispiel *A Song of Ice and Fire* von George R.R. Martin

Jeder Prozess arbeitet in einem Adressraum. Die Code- und Datenadressen, mit denen der Befehlsstrom auf den Speicher zugreift, gelten nur in diesem Adressraum. Ein Programm kann mehrere Befehlsströme enthalten, die alle im gleichen Adressraum ablaufen. Viele mehr oder weniger einfache Programme kommen aber mit einem Befehlsstrom aus. In den Urzeiten der Betriebssysteme konnte ein Programm ohnehin nur einen einzigen Befehlsstrom definieren. Jeder Prozess hatte seinen eigenen Adressraum. Es war nicht notwendig, die beiden Konzepte voneinander zu unterscheiden. Das ist der Grund, warum der Begriff „Prozess“ (engl.: *process*) mit verschiedenen Bedeutungen überladen ist.
Der folgende Abschnitt 8.1 bemüht sich zunächst um eine Begriffsklärung. Der Rest des Kapitels betrachtet, auf einer abstrakten Ebene, den Verlauf von Befehlsströmen und die damit einhergehenden Zustände eines Prozesses. Ein Befehlsstrom kann nicht nur aktiv sein, sondern auch auf eine Bedingung warten, anfangen, enden oder angehalten werden. Abschnitt 8.6 kombiniert alle Zustände und Übergänge in einem Diagramm. Abschnitt 8.7 fasst die wichtigsten Punkte dieses Kapitels zusammen. Konkreter wird es in Kapitel 12, welches Umschaltvorgänge, Bedingungen und die im Kern stattfindenden Zustandsänderungen der Prozesse beleuchtet. Kapitel 16 taucht noch tiefer ein und präzisiert den Umgang mit Registerinhalten beim Kernein- und -austritt.

Anmerkung: *Prozessoren schalten pro Sekunde vielfach zwischen den Prozessen um. Dadurch erscheint es einem Beobachter, als ob alle aktiven Prozesse gleichzeitig arbeiten. Der Aufwand für das Umschalten führt zu einem gewissen Leistungsverlust, den man aber für die Bequemlichkeit gerne in Kauf nimmt.*

8.1. Begriffsklärung

Um ein Programm auszuführen, muss das Betriebssystem eine ganze Reihe von Resourcen anlegen. Es braucht einen Adressraum, in dem der Code und die Daten des Programms liegen. Es braucht einen Prozess, der Rechenzeit für den Befehlsstrom erhält. Zum Prozess gehört ein Stapel, also ein weiterer Datenbereich im Adressraum. Mit der Programmiersprache C verbreitete sich die Konvention, dass einem Programm mindestens drei Datenströme für die Ein- und Ausgabe zur Verfügung stehen (`stdin`, `stdout`, `stderr`). In einer POSIX–Umgebung kommt dazu noch ein Satz von Signalen. Ein Betriebssystem kann seinen Anwendungen noch mehr zuordnen, zum Beispiel Fenster in einer graphischen Benutzeroberfläche. Zur Veranschaulichung genügen aber die oben aufgezählten Dinge. Gängige englische Begriffe für dieses Konglomerat von Resourcen eines laufenden Programms sind *process* oder *heavyweight process*.
Startet ein laufendes Programm in seinem eigenen Adressraum einen weiteren Befehlsstrom, braucht das Betriebssystem nur wenige zusätzliche Resourcen. Notwendig ist ein neuer Prozess mit einem eigenen Stapel im Adressraum. In einer POSIX–Umgebung gehört zum Prozess ein weiterer Satz von Signalen. Der Adressraum und die Datenströme existieren bereits. Ob weitere Resourcen mit dem neuen Befehlsstrom bzw. Prozess verbunden sind, hängt vom konkreten Betriebssystem ab. Geläufige englische Begriffe für einen solchen Befehlsstrom, ohne den Adressraum und die Datenströme, sind *thread* oder *lightweight process*.

Hier im Buch und in meiner Vorlesung verwende ich den deutschen Begriff „Prozess“ im leichtgewichtigen Sinne. Er bezeichnet ein Kernobjekt, das vom Kern Rechenzeit für einen Befehlsstrom erhält. Die passende englische Übersetzung ist *thread*, auch wenn ich den Satz von POSIX–Signalen als Anhängsel und nicht als Teil des Prozesses betrachte. Diese Sichtweise entspricht der Wettstein'schen Systemarchitektur.[1] Die beste englische Übersetzung für „Adressraum“ lautet *address space*. Für das gesamte Konglomerat, den *heavyweight process*, habe ich noch keinen deutschen Begriff gewählt. Vorschläge sind willkommen.

Nicht nur der Kern schaltet zwischen Befehlsströmen um. Ein Programm kann mehrere Befehlsströme definieren, zwischen denen es selbst hin- und herspringt. Aus Sicht des Kerns existiert für das Programm aber nur ein Prozess, der Rechenzeit erhält. Deshalb kann auch immer nur einer der Befehlsströme des Programms arbeiten, selbst wenn im Rechner mehrere Prozessoren stecken. Zur Unterscheidung der verschiedenen Spielarten von Befehlsströmen verwende ich hier die englischen Begriffe:

native threads, kernel–space threads**:** Befehlsströme, von denen der Kern weiß. Jeder *native thread* ist ein Prozess und bewirbt sich selbst um Rechenzeit.

user–space threads: Befehlsströme im gleichen Adressraum, zwischen denen eine Bibliothek umschaltet. Der Kern sieht nur einen Prozess, dem er Rechenzeit gibt. Eine Implementierung dieser Technik ist GNU Pth.

POSIX threads, Pthreads**:** Eine standardisierte API für Befehlsströme. Je nach Implementierung kommen *native threads* oder *user–space threads* zum Einsatz.

Die Dokumentation von GNU Pth enthält eine gute, kurze Einführung zu den Begriffen:
`https://www.gnu.org/software/pth/pth-manual.html#the_world_of_threading`

8.2. Aufgreifer

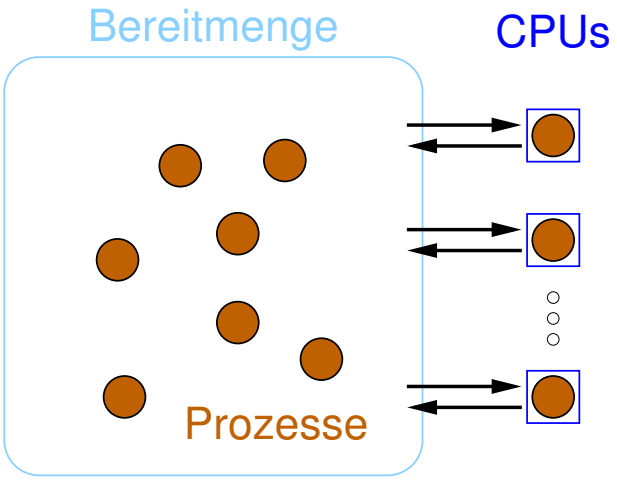


Abbildung 8.2.: Bereitmenge und Prozessoren

Der Kern eines Betriebssystems verfügt über eine begrenzte Anzahl von Prozessoren, um die Befehlsströme aller aktiven Prozesse voranzutreiben. Abbildung 8.2 zeigt die Situation. Die blauen Quadrate auf der rechten Seite stellen die Prozessoren (CPUs) dar. Auf jedem Prozessor läuft ein Prozess, eingezeichnet als brauner Kreis. In der *Bereitmenge* liegen weitere Prozesse, für die gerade kein Prozessor übrig ist.
Der *Aufgreifer* (engl.: *scheduler*) bestimmt, welcher Prozess auf welchem Prozessor wie lange rechnen darf. Zu bestimmten Zeitpunkten entscheidet er auf jeweils einem der Prozessoren, welcher Prozess aus der Bereitmenge diesen Prozessor übernimmt, oder ob der laufende Prozess noch weitermachen darf.
Alle Prozesse in der Verantwortung des Aufgreifers sind *aktiv*. Prozesse auf einer CPU befinden sich im Zustand *Rechnend*, die anderen im Zustand *Bereit*. Abbildung 8.3 zeigt diese beiden Zustände und die entsprechenden Übergänge, *aufgreifen* und *zurückstellen*. Wenn der Aufgreifer einen Prozess in die Bereitmenge zurückstellt, muss er einen anderen aufgreifen. Außerdem kommt es vor, dass rechnende Prozesse sich vom Aufgreifer verabschieden. Auch dann muss dieser einen anderen Prozess aufgreifen.
Damit in solchen Fällen immer mindestens ein Prozess in der Bereitmenge liegt, gibt es für jeden Prozessor einen *Leerlaufprozess* (engl.: *idle process*). Wenn im System wirklich gar nichts zu tun ist, dann „arbeiten“ alle Leerlaufprozesse und die Bereitmenge ist leer. Leerlaufprozesse haben die Aufgabe, möglichst stromsparend nichts zu tun. Der Aufgreifer kann sie dabei unterstützen, indem er zum Beispiel den Prozessor heruntertaktet.

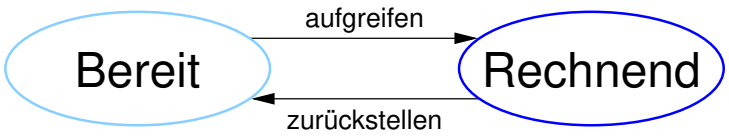


Abbildung 8.3.: Zustände aktiver Prozesse

8.3. Bedingungen

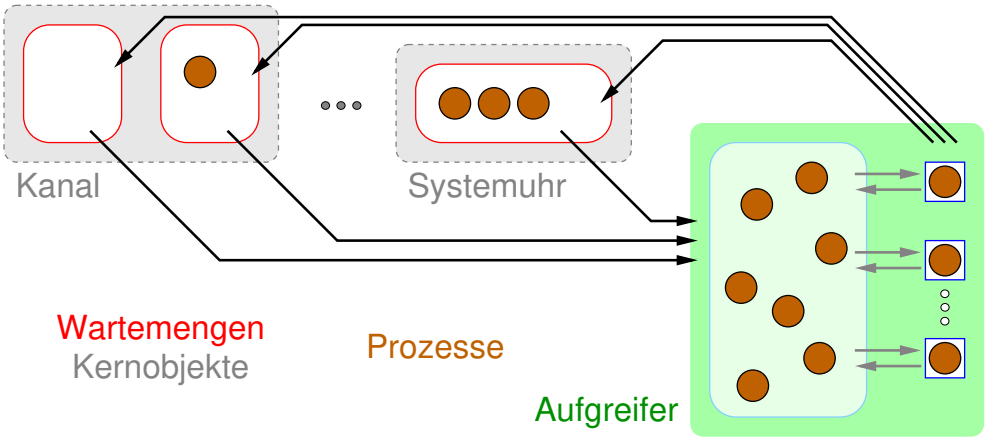


Abbildung 8.4.: Aufgreifer und Wartemengen

Ein rechnender Prozess kann auf Bedingungen stoßen, deren Erfüllung er abwarten muss. Zum Beispiel wenn er eine Benutzereingabe fordert, Daten von einer Festplatte benötigt, eine Nachricht von einem anderen Prozess erwartet oder wenn er eine wiederkehrende Aufgabe hat, die nur alle paar Sekunden zu erledigen ist. In solchen Fällen stoppt der Befehlsstrom vorübergehend, der Prozess *blockiert*.
Abbildung 8.4 zeigt die Situation. Auf der rechten Seite, grün unterlegt, befindet sich der Aufgreifer mit seiner Bereitmenge, den Prozessoren und allen aktiven Prozessen. Dieser Teil der Zeichnung ist aus Abbildung 8.2 bekannt. Stößt ein rechnender Prozess auf eine noch nicht erfüllte Bedingung, dann verlässt er den Bereich des Aufgreifers, der Prozessor wird frei. Der Aufgreifer holt für diesen Prozessor sofort einen neuen Prozess aus der Bereitmenge.
Oben in Abbildung 8.4 sind beispielhaft zwei Kernobjekte eingezeichnet, die verschiedene Bedingungen implementieren. An die Systemuhr wenden sich Prozesse, die für eine bestimmte Zeit schlafen wollen. Zum Beispiel weil sie nur all paar Minuten nach neu eingetroffenen Emails oder ähnlichem schauen. In der rot umrandeten *Wartemenge* (engl.: *queue*) liegen die schlafenden, also blockierten Prozesse nach dem Zeitpunkt des Aufweckens geordnet. So kann die Systemuhr schnell herausfinden, wieviel Zeit noch verstreichen muss, bis sie den nächsten Prozess weckt.

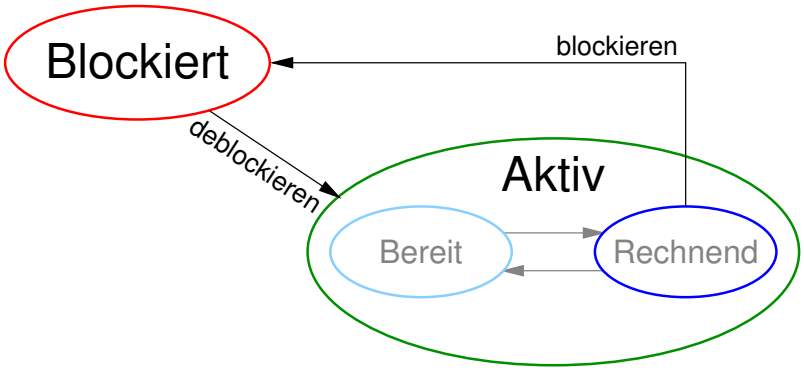


Abbildung 8.5.: Zustände beim Blockieren und Deblockieren

Bedingungen und die Ausgestaltung von Wartemengen ergeben sich aus den Operationen, die ein Kernobjekt implementiert. Die Kanäle aus Kapitel 2 bieten Operationen zum Senden und Empfangen an. Dass ein Empfänger warten muss, bis eine Nachricht gesendet wird, ist klar. Man kann aber auch eine Sendeoperation anbieten, die wartet, bis die gesendete Nachricht empfangen wird. Das ist eine andere Art von Bedingung, deshalb die zweite Wartemenge im Kanal aus Abbildung 8.4.
Abbildung 8.5 zeigt das Zustandsdiagramm mit dem neuen Zustand *Blockiert*. Die Zustände Bereit und Rechnend aus Abbildung 8.3 sind hier unter *Aktiv* zusammengefasst. Der Übergang *blockieren* führt einen rechnenden Prozess aus dem Bereich der aktiven Prozesse heraus. Ein blockierter Prozess ist dem Aufgreifer nicht mehr bekannt. Statt dessen liegt er in der Wartemenge für die Bedingung, die ihm fehlt. Sobald das zuständige Kernobjekt erkennt, dass die Bedingung erfüllt ist, *deblockiert* es den Prozess und übergibt ihn dem Aufgreifer. Das bringt den Prozess zurück in den Zustand Aktiv. Ob der deblockierte Prozess danach bereit steht oder gleich rechnet, entscheidet der Aufgreifer. Dieses Detail ist nicht mehr eingezeichnet, um die Darstellung nicht zu überfrachten.

Ein Prozess kann jeweils nur an einem Kernobjekt blockieren, also auch nur auf eine Bedingung warten.[2] Manchmal ist es aber praktisch, auf eine von mehreren Bedingungen zu warten. Zum Beispiel wenn eine Anwendung mehrere Eingangskanäle hat, aber nicht für jeden Kanal einen eigenen Empfangsprozess einsetzen will. Auch beim Schreiben in verschiedene Datenströme spart ein Prozess Zeit, wenn er zuerst dort schreibt, wo er nicht blockiert.
Für solche Anwendungszwecke bietet zum Beispiel POSIX eine Möglichkeit, mehrere Bedingungen zu kombinieren. Die Funktionen `select` bzw. `poll` arbeiten mit Datenströmen. Der aufrufende Prozess gibt an, von welchen Strömen er lesen und in welche er schreiben möchte. Er ruft aber damit weder Lese- noch Schreiboperationen auf. Der Aufruf blockiert den Prozess so lange, bis mindestens eine der gewünschten Operationen möglich ist, ohne zu blockieren. Beim Deblockieren erfährt der Prozess, welche Ströme gerade kommunikationsbereit sind. Dann führt er die entsprechenden Lese- oder Schreiboperationen durch, ohne zu blockieren. Anschließend passt er die Liste der Ströme an und wartet auf die nächste Gelegenheit.

[2] abgesehen von *Timeouts*, die eine Bedingung mit einer Wartezeit kombinieren

8.4. Anfang und Ende

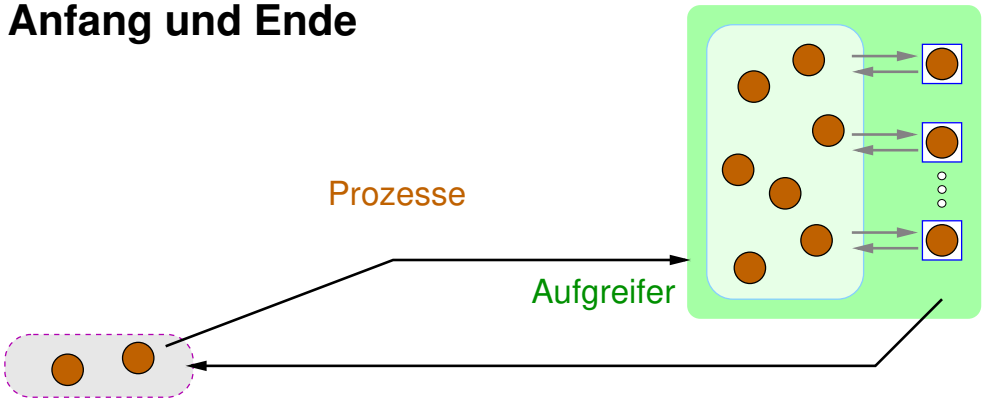


Abbildung 8.6.: Aktivieren und Enden

Zum Lebenslauf eines Prozesses gehört, dass er einmal erzeugt wird und irgendwann endet. Im guten Fall ist das Ende selbstbestimmt, weil der Befehlsstrom vollständig abgearbeitet wurde. Hierzu gehört auch, dass der Benutzer eine Anwendung verlässt oder dass Systemprozesse beim Herunterfahren des Rechners ihre Arbeit einstellen. In weniger guten Fällen werden Prozesse abgeschossen oder stürzen ab. Hardwareprobleme wie Stromausfälle betrachten wir hier nicht, da das Betriebssystem darauf keinen Einfluss hat.
Abbildung 8.6 zeigt auf der rechten Seite den inzwischen bekannten, grün unterlegten Bereich des Aufgreifers aus Abbildung 8.2. Links unten befindet sich ein neuer Bereich für *inaktive* Prozesse. Zu diesen gehören:

- neu erzeugte Prozesse, die noch nicht laufen
- beendete Prozesse, die noch herumliegen
- angehaltene Prozesse, siehe Abschnitt 8.5

Ein neu erzeugter Prozess ist zunächst inaktiv, bis man ihn dem Aufgreifer übergibt und damit aktiviert. Am Ende seines Befehlsstroms verlässt er den Bereich des Aufgreifers. Der Prozess ist fertig und bleibt inaktiv, bis er gelöscht wird.

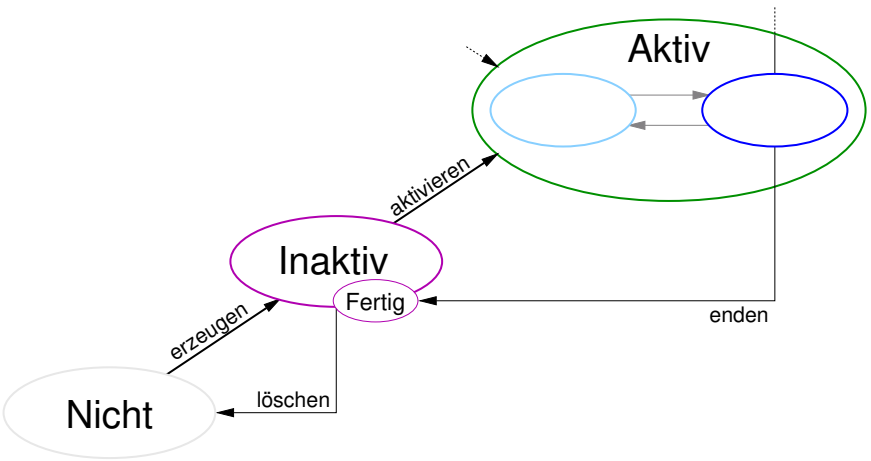


Abbildung 8.7.: Zustände beim Erzeugen und Enden

Das Zustandsdiagramm in Abbildung 8.7 knüpft an das aus Abbildung 8.5 an. Der neu eingeführte Start- und Endzustand ist *Nicht*. Prozesse, von denen das System nichts weiß, existieren nicht.[3] Ein neuer Prozess wird aus dem Nichts *erzeugt* und landet im Zustand *Inaktiv*. Beim Erzeugen entsteht ein Kernobjekt, welches den Prozess repräsentiert und ihm eine Identität verleiht. Der neue, inaktive Prozess ist weder dem Aufgreifer bekannt, noch liegt er in der Wartemenge für eine Bedingung. Ohne äußeren Anstoß erhält er keine Rechenzeit. Der Erzeuger kann so Initialisierungen vornehmen, zum Beispiel eine Priorität setzen. Anschließend *aktiviert* er den Prozess. Der aktive Prozess erhält nun Rechenzeit vom Aufgreifer und kann an Bedingungen blockieren, wie weiter oben beschrieben. Diese Vorgänge sind in Abbildung 8.7 nur noch angedeutet.
Ein Prozess *endet* mit einem Kernaufruf, der ihn in den Zustand *Fertig* versetzt. Dieser Kernaufruf bildet den Abschluss des Befehlsstroms. Der Prozess ist nun wieder inaktiv, lässt sich aber nicht mehr aktivieren. Deshalb ist Fertig im Zustandsdiagramm ein Sonderfall von Inaktiv. Der Erzeuger oder andere können den fertigen Prozess noch inspizieren, um Ergebnisse zu erhalten oder Statistiken zu sammeln. Letztendlich *löscht* das System den Prozess und sein Kernobjekt, er verschwindet im Nichts.

Anmerkung: *„Nicht“ ist kein Zustand eines bestimmten Prozesses. Löschen gefolgt von Erzeugen liefert einen neuen Prozess. Es stellt nicht den gelöschten Prozess wieder her.*

In den Zustand Fertig gelangt ein Prozess selbstbestimmt, indem er den letzten Befehl seines Befehlsstroms ausführt. Das geht nur aus dem Zustand Rechnend heraus, so wie auch nur ein rechnender Prozess an einer Bedingung blockieren kann. Ein fremdbestimmtes Abschießen führt immer nach Inaktiv, egal ob der Prozess anschließend wieder aktiviert werden kann oder nicht. Diese Übergänge, die den Prozess *deaktivieren*, sind Thema von Abschnitt 8.5. Löschen kann man alle inaktiven Prozesse, ob fertig oder nicht.
Das Ende eines Prozesses ist eine Bedingung, auf die insbesondere der Erzeuger warten kann. Mit seinem letzten Kernaufruf übergibt ein Prozess noch einen Ergebniswert, den wartende Prozesse erhalten. Falls niemand auf das Ende wartet, muss das System entscheiden, ob es das Ergebnis aufbewahrt oder den Prozess mitsamt Ergebnis löscht. Im Unix–Umfeld nennt man einen fertigen Prozess, dessen Ergebnis niemand abholt, *Zombie*. Stirbt ein Prozess durch Abschuss, liefert er selbst kein Ergebnis. Darauf wartende Prozesse deblockieren dann mit einem Fehler.

[3] noch nicht, nicht mehr oder niemals nicht

8.5. Anhalten

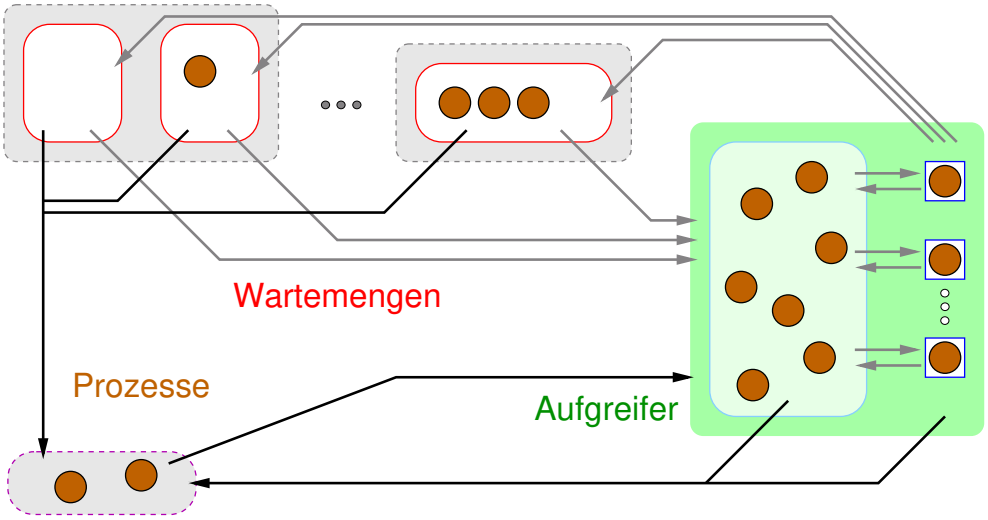


Abbildung 8.8.: Aktivieren und Deaktivieren

Abbildung 8.8 zeigt das vollständige Mengenbild zu den Prozesszuständen. Rechts, im grün unterlegten Bereich des Aufgreifers, tummeln sich die aktiven Prozesse. Oben, in den Wartemengen von Kernobjekten, stecken blockierte Prozesse. Links unten liegen inaktive Prozesse. Diese Bereiche sind bereits aus den vorhergehenden Abschnitten bekannt. Neu hinzugekommen sind hier lediglich einige Übergänge nach links unten. Man *deaktiviert* einen aktiven oder blockierten Prozess, um ihn inaktiv zu machen. Das kann ein vorübergehendes Anhalten oder ein endgültiger Abschuss sein.
Abbildung 8.9 ergänzt die Übergänge des Zustandsdiagramms. Das Deaktivieren eines aktiven Prozesses macht ihn inaktiv, aber nicht fertig. Auch der Übergang von Blockiert zu Inaktiv heißt deaktivieren. Beim Deaktivieren eines Prozesses muss man also auf dessen aktuellen Zustand keine Rücksicht nehmen. Es ist Aufgabe des Kerns, die jeweils notwendigen Aktionen durchzuführen. Ob sich der Prozess anschließend wieder aktivieren lässt, hängt von den Umständen des Deaktivierens ab.

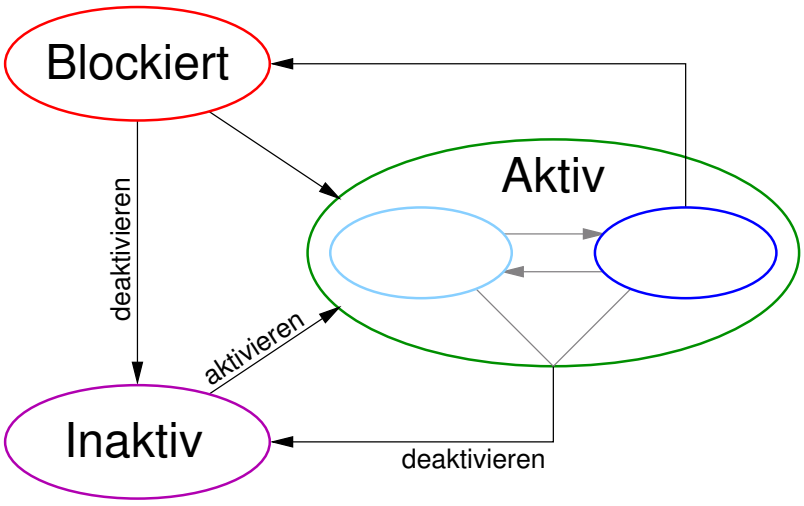


Abbildung 8.9.: Zustände beim Deaktivieren

Wird ein aktiver Prozess vom System deaktiviert, zum Beispiel wegen eines Programmfehlers, kann der Befehlsstrom nicht weiterlaufen. Dann ist höchstens die Fortsetzung in einer Fehlerbehandlungsroutine möglich, falls eine solche hinterlegt ist. Wird ein aktiver Prozess dagegen vom Benutzer deaktiviert, kann dieser ihn anschließend wieder aktivieren. Für den Befehlsstrom ist das nicht anders, als hätte der Prozess längere Zeit in der Bereitmenge gelegen, ohne einen Prozessor zu bekommen. So arbeiten auch *Break Points* (dt.: Stoppstellen) und der Einzelschrittmodus in einem Debugger.
Wird ein blockierter Prozess deaktiviert, ist das Aktivieren ein Problem. Der Prozess hat auf eine Bedingung gewartet, die zum Zeitpunkt des Deaktivierens nicht erfüllt war. Aktiviert man den Prozess, obwohl die Bedingung nicht erfüllt ist, dann läuft der Befehlsstrom unter falschen Voraussetzungen weiter und berechnet wahrscheinlich Blödsinn. Ein Reaktivieren in den Zustand *Blockiert* sieht das Diagramm in Abbildung 8.9 aber nicht vor. Eine Möglichkeit ist, den Kernaufruf des Befehlsstroms mit einem Fehler abzubrechen und auf die Fehlerbehandlung des Programms zu hoffen.
Tatsächlich handelt es sich dabei um eine Einschränkung des vereinfachten Prozessmodells, das ich hier vorstelle. Ein inaktiver Prozess befindet sich in einem Ruhezustand, in dem er weder dem Aufgreifer noch einer Wartemenge bekannt ist. Aus diesem Zustand kann man ihn problemlos *löschen*. Aber das Deaktivieren eines blockierten Prozesses reißt ihn aus einer Wartemenge, was den blockierten Kernaufruf unterbricht. Ohne dieses harte Anhalten könnte man blockierte Prozesse nicht abschießen.[4]

Anmerkung: *Für ein weiches Anhalten und Fortsetzen (engl.: suspend, resume) wie in POSIX braucht man einen weiteren Prozesszustand. Ein blockiert–inaktiver Prozess bleibt in der Wartemenge. Falls sich die Bedingung erfüllt, wird er inaktiv statt zu deblockieren.*

[4] `https://lwn.net/Articles/288056/` Drei Varianten des Blockierens in Linux

8.6. Zustandsdiagramm

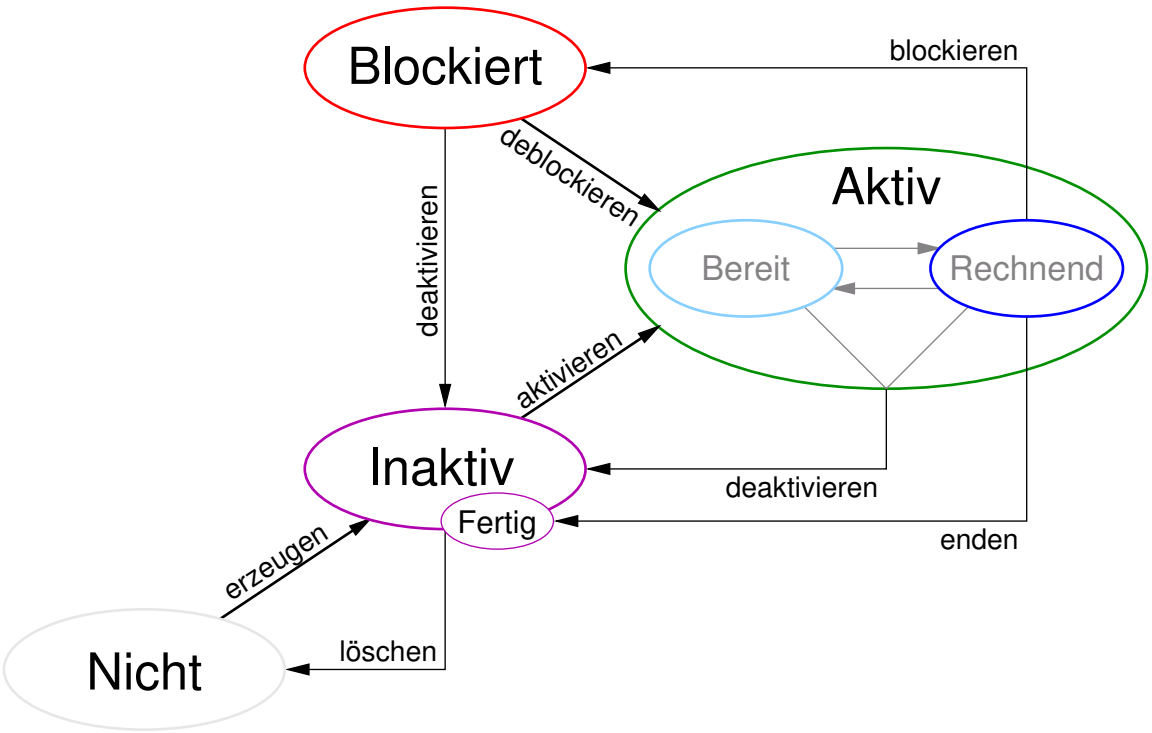


Abbildung 8.10.: Gesamtbild der Prozesszustände

Abbildung 8.10 zeigt das vollständige Diagramm der Prozesszustände aus den vorhergehenden Abschnitten. Ein Prozess entsteht links unten. Er wird aus dem Nichts erzeugt und anschließend aktiviert. Aktive Prozesse rechnen entweder auf einem Prozessor, oder sie stehen bereit falls einer frei wird. Der Aufgreifer (engl.: *scheduler*) entscheidet, welcher aktive Prozess auf welchem Prozessor rechnet. Nur im Zustand Rechnend kommt der Befehlsstrom eines Prozesses voran. Ein rechnender Prozess kann enden, dann ist er fertig und wieder inaktiv. Mit dem Löschen verschwindet der Prozess aus dem System.
Ein rechnender Prozess kann in seinem Befehlsstrom auf Bedingungen stoßen, die nicht erfüllt sind. Dann blockiert der Prozess und der Befehlsstrom steht still. Sobald die Bedingung erfüllt ist, wird der Prozess deblockiert und kann wieder Rechenzeit erhalten. Im Gegensatz zum Blockieren ist das Deaktivieren nicht an den Befehlsstrom gebunden. Die Aktion erfolgt von außen, zum Beispiel beim Abschießen. Sie zwingt einen Prozess in den Zustand Inaktiv, egal was er gerade tut oder worauf er wartet. Ein erneutes Aktivieren kann klappen, muss es aber nicht.
Diagramme mit Prozesszuständen sind eine Hilfe, um die Zustände und Änderungen in einem System einzuordnen und in einen Zusammenhang zu setzen. Das Zustandsdiagramm in Abbildung 8.10 entspricht dem Detailgrad, den ich hier im Buch anstrebe. Konkrete Betriebssysteme können weitere Zustände unterscheiden. Andere Prozessmodelle verwenden weniger Zustände und verzichten lieber auf Details wie das Deaktivieren. Es folgen einige Beispiele.

8.7. Knackpunkte

- Der Begriff „Prozess“ hier entspricht ungefähr dem englischen *thread*, nicht *process*. Zum englischen *process* gehört für gewöhnlich ein eigener Adressraum.
- Prozesse sind Objekte, die der Kern verwalten muss. Zum Beispiel, indem er sie in verschiedene Mengen steckt.
- Prozesse repräsentieren Befehlsströme. Ein Befehlsstrom braucht Rechenzeit, also einen Prozessor, um voranzukommen.
- Der Aufgreifer vergibt Rechenzeit an aktive Prozesse. Ein aktiver Prozess ist entweder rechnend oder bereit.
- Ein rechnender Prozess hat gerade einen Prozessor.
- Ein bereiter Prozess liegt in der Bereitmenge, ohne einen Prozessor.
- Nur ein rechnender Prozess kann auf eine Stelle im Befehlsstrom stoßen, wo er auf etwas warten muss. Der Prozess blockiert und landet in der passenden Wartemenge.
- Blockierte Prozesse holt der Kern aus der Wartemenge und deblockiert sie, sobald etwas passiert ist, worauf sie gewartet haben. Deblockierte Prozesse landen wieder beim Aufgreifer.
- Die drei Zustände Bereit, Rechnend und Blockiert bilden den kleinsten, gemeinsamen Nenner aller Prozesszustandsmodelle. Es gibt viele solche Modelle, mit unterschiedlichem Detailgrad, mehr oder weniger speziell für konkrete Betriebssysteme.
- Das hier vorgestellte Modell entspricht dem Detailgrad, den ich früher in meiner Vorlesung durchgenommen habe. Es repräsentiert kein konkretes Betriebssystem. Deren Modelle haben mehr Zustände.
- Nur ein rechnender Prozess kann fertig werden, also das Ende seines Befehlsstroms erreichen. Dann ist er inaktiv, ohne Chance auf Fortsetzung.
- Ein neu erzeugter Prozess ist noch inaktiv, bis man ihn aktiviert, also startet.
- Prozesse können außerhalb ihres normalen Ablaufs deaktiviert werden, zum Beispiel durch Debugger oder Abschuss. In manchen Fällen kann man sie anschließend auch wieder aktivieren, also weiterlaufen lassen.
- Der Start- und Endzustand „Nicht“ ist ein Kniff, um die Übergänge für das Erzeugen und Löschen von Prozessen in das Diagramm zu integrieren.

9. Synchronisation

Prozesse führen sequentielle Befehlsströme aus. Der ausgeführte Code und seine Eingabedaten bestimmen die Reihenfolge der Befehle in jedem Strom. Aktive Prozesse erhalten Rechenzeit und arbeiten unabhängig voneinander, teils gleichzeitig auf verschiedenen Prozessoren, teils abwechselnd auf dem gleichen Prozessor. In welcher Reihenfolge das System Befehle aus verschiedenen Strömen ausführt, lässt sich im Allgemeinen nicht vorhersagen. Bei unabhängigen Prozessen ist das kein Problem. Aber wenn Prozesse miteinander arbeiten sollen, braucht man eine gewisse Kontrolle über die Reihenfolge ihrer Befehle. Zum Beispiel:

- Schritt A in Prozess X muss erledigt sein, bevor Schritt B in Prozess Y beginnt.
- Prozesse U und V dürfen nicht gleichzeitig in Datenstruktur D schreiben.

Die Prozesse müssen deshalb ihren Fortschritt miteinander abstimmen, sich synchronisieren. Das bedeutet, dass immer mal wieder ein Prozess auf einen anderen wartet. Wie das Warten funktioniert, ist aus den vorhergehenden Kapiteln bekannt. Die betroffenen Prozesse blockieren und landen in einer Wartemenge, bis sie wieder deblockiert werden. Das sind allerdings interne Aktionen des Kerns. Prozesse in Instanzen können sie nicht direkt aufrufen.
Abschnitt 9.1 führt Interaktionsobjekte ein, als Schnittstelle des Kerns zu den Instanzen. Interaktion ist ein Oberbegriff, der unter anderem Synchronisation umfasst. Abschnitt 9.2 beschreibt das Semaphor, ein einfaches und weit verbreitetes Synchronisationsobjekt. Abschnitt 9.3 erklärt das Problem kritischer Abschnitte, die man durch Synchronisation schützen muss. Abschnitt 9.4 geht auf verschiedene Verwendungsmöglichkeiten für Semaphore ein. Abschnitt 9.5 fasst die wichtigsten Punkte dieses Kapitels zusammen. In Kapitel 10 folgen weitere Interaktionsobjekte zur Synchronisation. Aus Kapitel 2 ist bereits bekannt, wie man Interaktionsobjekte zur Implementierung von Interaktionsmustern verwendet.

9.1. Interaktionsobjekte

Der Kern bietet Interaktionsobjekte an, mit denen Prozesse ihre Beziehungen strukturieren. Die Methoden der Interaktionsobjekte erkennen, ob ein aufrufender Prozess warten muss und wann ein wartender Prozess fortsetzen kann. Die Interaktionsobjekte im Kern verwalten Wartemengen, sie lösen das Blockieren und Deblockieren von Prozessen aus.

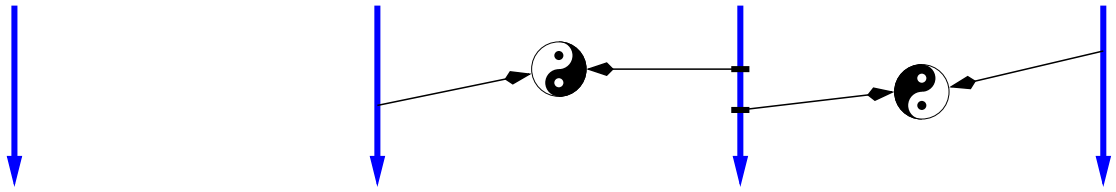


Abbildung 9.1.: Prozesse rufen Methoden an Interaktionsobjekten auf

Abbildung 9.1 veranschaulicht die Rolle von Interaktionsobjekten. Die blauen Pfeile stehen für Prozesse, also langlebige Befehlsströme, ähnlich wie in Abbildung 12.1 auf Seite 123. Es gibt keine gemeinsame Zeitachse. Alle Prozesse kommen entlang ihrer Pfeile voran, aber mit variabler Geschwindigkeit. Dazwischen sind zwei Interaktionsobjekte eingezeichnet, vergleichbar mit Abbildung 7.4 auf Seite 74. Aufrufpfeile führen von Prozessen zu Interaktionsobjekten. Ein Interaktionsereignis tritt dann ein, wenn an einem Objekt jeweils ein Aufruf der schwarzen und der weißen Seite zueinander finden.
Zur Synchronisation koppelt einer der Prozess seinen Ablauf an ein Interaktionsereignis. Im Beispiel geschieht das bei der schwarzen Methode. Falls nötig blockiert diese den aufrufenden Prozess bis zum Ereignis, deshalb der Querbalken am Anfang der Aufrufpfeile. Das ist die *synchrone* Kopplung. Die weiße Methode dagegen arbeitet *asynchron*. Der aufrufende Prozess läuft weiter, egal ob das Interaktionsereignis gerade stattfand oder noch nicht.
Die Synchronisation ermöglicht Aussagen über den relativen Fortschritt verschiedener Prozesse. Drei der vier Prozesse in Abbildung 9.1 laufen ohne Verzögerung, so schnell sie können. Der Prozess zwischen den beiden Interaktionsobjekten aber nicht, denn er ruft zwei synchrone Methoden auf. Durch diese koordiniert er seinen Ablauf mit dem der beiden Nachbarprozesse. Vor dem ersten Aufruf ist er ebenfalls unabhängig von den anderen. An der Aufrufstelle kommt er aber erst vorbei, wenn sein linker Nachbar die weiße Methode aufruft. Das kann allerdings schon vor dem Aufruf der schwarzen Methode passiert sein. An der zweiten Aufrufstelle bleibt der mittlere Prozess wiederum hängen, falls der rechte Nachbar die weiße Methode noch nicht aufgerufen hat. In den Bereich nach der zweiten Aufrufstelle kommt er also erst, nachdem beide Nachbarprozesse ihre Beiträge zu den Interaktionsereignissen liefern.

9.2. Semaphor

Edsger W. Dijkstra führte Semaphore als Mittel zur Synchronisation von Prozessen ein.[1] Ein Semaphor ist ein Zähler mit Operationen zum Hoch- und Runterzählen. Der Zähler wird nie negativ. Runterzählen bei 0 blockiert den Aufrufer. Ist beim Hochzählen mindestens ein Runterzähler blockiert, darf einer von ihnen fortsetzen. Der Zählerstand bleibt dabei 0, weil sich das aufgerufene Hochzählen und das deblockierte Runterzählen ausgleichen. Ist bei einem Zählerstand größer 0 ein Runterzähler blockiert, dann liegt ein Programmierfehler im Semaphor vor. Kernobjekte dürfen Prozesse nur so lange blockieren, bis die erwartete Bedingung erfüllt ist, das erwartete Ereignis stattfinden kann. Das Interaktionsereignis bei einem Semaphor ist das Runterzählen.

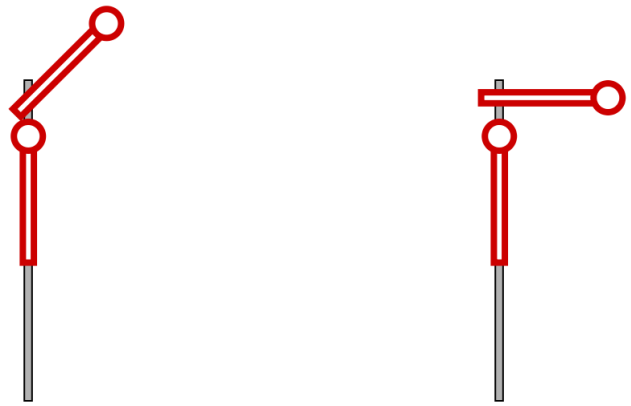


Abbildung 9.2.: Eisenbahnsignale
Quelle: Public Domain, über Wikimedia Commons
(Hp1_Form.svg, Hp0_Form.svg)

Anmerkung: *Der Begriff* Semaphor *stammt aus der Eisenbahntechnik. Es ist ein Signal, das in seiner einfachsten Form nur die zwei Zustände in Abbildung 9.2 annimmt. Links steht der Flügel diagonal, der Gleisabschnitt dahinter ist frei. Ein Zug darf passieren. Der Flügel wechselt dabei in die horizontale Position rechts. Der nächste Zug muss warten, bis der vorhergehende den Gleisabschnitt verlässt.*

Dijkstra nannte die beiden Operationen des Semaphors `P` zum Runterzählen und `V` zum Hochzählen. `P` ist synchron, `V` asynchron. Die Buchstaben beziehen sich auf niederländische Begriffe. Man kann sie zum Beispiel mit *passieren* und *freigeben* assoziieren. Reale Implementierungen verwenden oft andere Namen und bieten auch weitere Methoden an. Zum Verständnis der Funktionsweise genügen aber diese beiden.
Abbildung 9.3 beschreibt den Aufbau eines Semaphors und den Ablauf der beiden Operationen. Als Attribute braucht das Semaphor einen Zähler und eine Wartemenge (engl.: *queue*). Die Dateninvariante schließt den eingangs erwähnten, ungültigen Fall aus: Kein Prozess darf warten, wenn der Zähler über 0 steht. Alle Methoden müssen dafür sorgen,

[1] Edsger W. Dijkstra: *Cooperating sequential processes*, 1965
`https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`

Semaphor

Attribute:

Zähler, nicht negativ. Initialwert beim Erzeugen anzugeben.

Wartemenge für Prozesse. Initial leer.

Dateninvariante:

Wenn **Zähler** > 0, dann **Wartemenge** leer.

Methoden:

`P` — *Runterzählen, synchron*

- **Zähler** > 0 ?

ja: **Zähler** vermindern

nein: Aufrufer blockieren und in **Wartemenge** stecken

`V` — *Hochzählen, asynchron*

- **Wartemenge** leer?

ja: **Zähler** erhöhen

nein: Prozess aus **Wartemenge** nehmen und deblockieren

Abbildung 9.3.: Attribute und Abläufe eines Semaphors

dass die Dateninvariante erfüllt bleibt. Die bei den Methoden stehenden Abläufe beschreiben jeweils eine Kernaktion, die durch den jeweiligen Kernaufruf ausgelöst wird. Diese Abläufe können also selbst nicht blockieren.
`P` beginnt mit einer Fallunterscheidung. Entweder löst der Aufruf sofort das vom Aufrufer erwartete Interaktionsereignis aus oder nicht. Wenn der Zähler größer 0 ist, wird sein Stand sofort vermindert und der Aufrufer darf weiterarbeiten. Damit ist sowohl die Kernaktion als auch die aufgerufene Kernoperation abgeschlossen. Anderenfalls kann das Runterzählen, also das Ereignis, erst später stattfinden. Die Operation ist synchron, deshalb muss der aufrufende Prozess bis dahin blockiert werden. Genau das geschieht in der Kernaktion. Der blockierte Prozess landet in der Wartemenge, damit das Kernobjekt ihn später wiederfindet, um die Kernoperation abzuschließen. Nach dem Eintragen in die Wartemenge endet die Kernaktion.
Auch `V` beginnt mit einer Fallunterscheidung, ob der Aufruf sofort ein Interaktionsereignis auslöst oder nicht. Das Ereignis betrifft hier aber nicht den Aufrufer, sondern einen der blockierten Prozesse. Falls kein Prozess in der Wartemenge liegt, entsteht kein Ereignis. Dann erhöht die Kernaktion einfach den Zähler und ist fertig. Liegt ein Prozess in der Wartemenge, würde das Erhöhen des Zählers zum ungültigen Fall führen und die Dateninvariante verletzen. Stattdessen nimmt die Methode einen der Prozesse aus der Wartemenge und deblockiert ihn. Der Zählerstand bleibt unverändert, weil sich die Effekte des aufgerufenen `V` und des deblockierten `P` aufheben. Trotzdem fand das Interaktionsereignis statt, auf das der deblockierte Prozess wartete. Dessen Kernoperation ist nun abgeschlossen, ebenso wie die des Aufrufers.
Die Dateninvariante muss nur am Ende jeder Kernaktion gelten, aber nicht im Verlauf. Der Ablauf von `V` könnte deshalb auch so aussehen:

- Zähler erhöhen
- falls Wartemenge nicht leer:
  - Prozess aus Wartemenge nehmen und deblockieren
  - Zähler vermindern

Hier wird deutlicher, wie sich Hoch- und Runterzählen aufheben. Außerdem zeigt dieser Ablauf, dass das Erkennen eines Interaktionsereignisses nicht nur am Anfang einer Methode stehen kann. Ich bevorzuge allerdings die Variante aus Abbildung 9.3, welche keine Zeit für unnötige Änderungen des Zählerstands verschwendet.
Die Wartemenge ist hier als abstrakter Datentyp zu verstehen. Die Kernaktionen können Prozesse hineinstecken, herausnehmen sowie prüfen, ob die Menge leer ist oder nicht. Warten mehrere Prozesse, spielt es für die Funktion des Semaphors keine Rolle, welcher aus der Menge entnommen wird. Für eine konkrete Implementierung wählt man eine Strategie, zum Beispiel nach der Reihenfolge des Einfügens oder anhand der Prioritäten der wartenden Prozesse. Auch die zufällige Auswahl wäre eine Strategie, aber keine sinnvolle. Mit ihr könnte man nicht einschätzen, wie sich das System unter Last verhält, also wenn meist mehrere Prozesse in der Wartemenge liegen.
Dass `V` asynchron arbeitet bedeutet, dass die Operation den Aufrufer nicht blockiert. Trotzdem kann er den Prozessor an den deblockierten Prozess verlieren. Das ist kein Blockieren, denn der Aufrufer bleibt im Prozesszustand Aktiv aus Abschnitt 8.3. Er erhält also bei nächster Gelegenheit wieder Rechenzeit vom Aufgreifer.

9.3. Kritische Abschnitte

Laufen die Kernaktionen `P` und `V` eines Semaphors gleichzeitig auf zwei verschiedenen Prozessoren ab, führt das zu Problemen. Zum Beispiel könnte `P` einen Zählerstand 0 vorfinden und `V` eine leere Wartemenge. Dann blockiert `P` den Aufrufer und fügt ihn in die Wartemenge ein. Gleichzeitig erhöht `V` den Zähler auf 1. Damit ist der ungültige Fall eingetreten, die Dateninvariante verletzt. Das darf nicht passieren.
Auch die gleichzeitige Ausführung anderer Kombinationen führt zu Problemen. Zweimal `P` beim Zählerstand 1 lässt zwei Prozesse passieren statt nur einen. Ob der Zähler danach auf 0, -1 oder einem sehr hohen, positiven Wert steht, hängt vom verwendeten Datentyp und dem genauen zeitlichen Ablauf des Verminderns ab. Das gleichzeitige Vermindern bei einem höheren Zählerstand kann zu einem falschen Ergebnis führen. Bei zweimal `V` können beide Aktionen versuchen, den gleichen Prozess aus der Wartemenge zu nehmen und zu deblockieren. Das gleichzeitige Hochzählen kann ebenso schiefgehen wie das Runterzählen.
Diese Problematik beschränkt sich nicht auf das Semaphor, und auch nicht nur auf Interaktionsobjekte. Überall wo mehr als zwei Prozesse unkoordiniert auf eine gemeinsame Datenstruktur zugreifen und mindestens einer schreibt, kommt es zu sogenannten *race conditions* (dt.: Wettlaufsituationen). Das führt dazu, dass zu nicht vorhersagbaren Zeitpunkten Prozesse eine inkonsistente Datenstruktur vorfinden oder hinterlassen. Dadurch arbeiten die Prozesse fehlerhaft, die Datenstruktur ist unbrauchbar.
Eine Ausnahme liegt vor, wenn höchstens ein Prozess schreibt und die Datenstruktur aus einem einzigen Speicherwort besteht, welches vom Prozessor *atomar* gelesen und geschrieben wird. Dann finden die lesenden Prozesse immer einen konsistenten Wert vor, entweder den alten oder den neuen. Schon wenn der Wert über zwei Speicherworte verteilt ist, könnte das Lesen eine Kombination aus einem alten und einem neuen Speicherwort liefern. Auch falls mehr als ein Prozess atomar schreibt entstehen Probleme, weil das Ändern eines Wertes mehrere Schritte erfordert:

1. alten Wert aus dem Speicher in ein Register lesen
2. Wert im Register verändern
3. neuen Wert in den Speicher schreiben

Lesen zwei Prozesse gleichzeitig den alten Wert aus dem Speicher und berechnen einen neuen, dann überschreibt einer der beiden die Änderung des anderen. So entsteht die oben beschriebene Situation, in der zwei `P` den Zählerstand von 1 vermindern und den Zähler dabei auf 0 setzen.

Ein *kritischer Abschnitt* im Code ist eine Folge von Befehlen, die *atomar* ablaufen muss. Sobald ein kritischer Abschnitt beginnt, und bis seine Ausführung, endet dürfen andere Befehlsströme nicht auf Speicherinhalte zugreifen, die der kritische Abschnitt benutzt. Weder lesend, weil der Speicherinhalt inkonsistent sein könnte, noch schreibend, weil der kritische Abschnitt nicht mit Änderungen rechnet.
Kritische Abschnitte beziehen sich also auf bestimmte Speicherinhalte. Zum Beispiel auf die Attribute eines Semaphors, oder auf eine beliebige andere Datenstruktur. Ein Prozess, der nur mit privaten Daten auf seinem Stapel arbeitet, hat keine kritischen Abschnitte. Ein Prozess, der alleine in seinem Adressraum arbeitet, ebenfalls nicht. Zwei Prozesse,

die im gleichen Adressraum jeweils mit privaten Daten auf ihrem Stapel und der Halde arbeiten, haben kritische Abschnitte. Denn beim Anfordern und Freigeben von Speicher auf der Halde greifen beide Befehlsströme auf die Verwaltungsdaten der Halde zu. Falls die Bibliothek zur Haldenverwaltung ihre kritischen Abschnitte schützt, können die Prozesse ansonsten ungestört nebeneinander arbeiten.
Zum Schutz kritischer Abschnitte dient oft eine Exklusivsperre, die zu Beginn gesetzt und am Ende wieder freigegeben wird. Das stellt sicher, dass höchstens ein kritischer Abschnitt auf die gesperrte Datenstruktur zugreift. Anders ausgedrückt serialisiert dieses Vorgehen die Ausführung aller kritischen Abschnitte, die die gleiche Sperre verwenden. Liegen die Datenstruktur und kritischen Abschnitte in einer Instanz, kann die Sperre ein Kernobjekt sein, zum Beispiel ein Semaphor. Initialisiert mit 1, gesperrt durch Aufruf von `P` und freigegeben mit `V`, funktioniert es als Exklusivsperre. Ein anderes Kernobjekt zu diesem Zweck ist das Mutex, das ich in Abschnitt 10.1 vorstelle.
Liegt die Datenstruktur im Kern und die kritischen Abschnitte in Kernaktionen, die nicht blockieren können, setzt man spezielle Prozessorbefehle ein. Diese stellen sicher, dass eine Lese- und eine Schreiboperation auf den Hauptspeicher atomar erfolgen, also ohne dass ein anderer Prozessor den Inhalt der gelesenen Speicherstelle ändert, bevor sie geschrieben wurde. So kann ein Befehlsstrom eine Speicherstelle auf „gesperrt“ setzen und dabei prüfen, dass sie vorher noch nicht auf diesem Wert stand. Falls doch gehört die Sperre einem anderen Befehlsstrom und der neue muss es in einer Schleife immer wieder probieren, bis er sie selbst erfolgreich setzen kann.
Solche Befehle kann man prinzipiell auch in Instanzen verwenden. Allerdings sollten Prozesse nach einem Fehlschlag blockieren, statt mit aktivem Warten (engl.: *polling*) Rechenzeit zu verschwenden. Umgekehrt muss das Freigeben der Sperre dann auch erkennen, dass ein Prozess blockierte und nun deblockiert werden muss. So spart man teure Kernaufrufe, wenn das Setzen der Sperre unmittelbar klappt und zwischendurch kein anderer Prozess blockiert.[2]

Achtung: *Alle Methoden der Interaktionsobjekte sind kritische Abschnitte. Bei allen Ablaufbeschreibungen setze ich voraus, dass ein Mechanismus im Kern die Ausführung dieser Kernaktionen serialisiert.*

[2] *futex — fast user-space locking*, abgerufen 2022-01-25
`https://man7.org/linux/man-pages/man7/futex.7.html`

9.4. Semaphore im Einsatz

Semaphore sind vielseitig verwendbar. Das *Little Book of Semaphores* [2] beleuchtet die Einsatzmöglichkeiten mit einer großen Sammlung von Synchronisationsproblemen. Teils handelt es sich um Beispiele, teils um Übungsaufgaben mit Lösungen. Die Problemstellungen sind mal mehr, mal weniger sinnvoll, aber durchweg unterhaltsam.
Die Verwendung als Exklusivsperre habe ich schon oben in Abschnitt 9.3 beschrieben. Abbildung 9.4 veranschaulicht diese Möglichkeit mit einem Interaktionsdiagramm. Die vier dargestellten Befehlsströme enthalten jeweils einen roten, kritischen Abschnitt. Das Semaphor in der Mitte ist als gelbes Fünfeck dargestellt, aufgeteilt in ein Rechteck für `P` und ein Dreieck für `V`. Die 1 darüber gibt den initialen Zählerstand des Semaphors an. Am Anfang jedes kritischen Abschnitts steht ein synchroner Aufruf von `P`, am Ende ein asynchroner von `V`. Wegen der Initialisierung mit 1 kann höchstens einer der Prozesse in seinem kritischen Abschnitt sein. Da der Zähler nur die Werte 0 oder 1 annimmt, spricht man bei dieser Verwendung von einem Binärsemaphor.

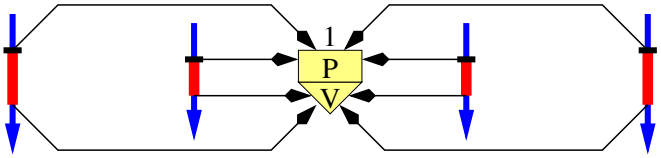


Abbildung 9.4.: Semaphor als Sperre

Eine zweite, ähnlich gelagerte Möglichkeit ist der Einsatz eines Semaphors als Teil einer Betriebsmittelverwaltung. Legt eine Instanz zum Beispiel einen Pool mit p Puffern an, so kann sie zusätzlich ein Semaphor mit dem Zählerstand p initialisieren. Beim Anfordern eines Puffers rufen Prozesse `P` auf. So stellen sie sicher, dass ein Puffer frei ist, bevor sie danach suchen. Beim Freigeben rufen sie `V`, um gegebenenfalls einen anderen Prozess zu deblockieren, der auf einen freien Puffer wartet. Hier arbeitet das Semaphor also ebenfalls als Sperre, wenn auch nicht als Exklusivsperre.

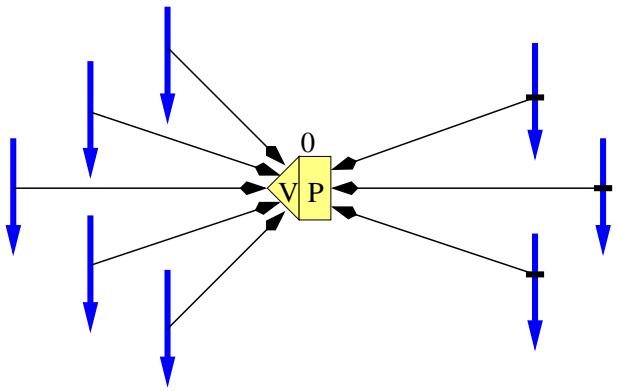


Abbildung 9.5.: Semaphor zum Signalisieren

Eine anders gelagerte Verwendung ist das Signalisieren, in Abbildung 9.5 dargestellt. Hier initialisiert man das Semaphor typischerweise mit 0, da zu Beginn noch kein Signal vor-

liegt. Prozesse rufen `V` auf, um ein Signal zu geben, oder `P`, um ein Signal abzuwarten. Ein Signal könnte zum Beispiel anzeigen, dass ein Prozess Daten zur Verarbeitung bereitgestellt hat. Verarbeitende Prozesse warten dann auf solche Signale.
Für bestimmte Einsatzzwecke, die das Semaphor abdeckt, kann ein Kern oder eine Bibliothek auch andere Objekttypen anbieten, deren Operationen sprechendere Namen tragen. In den folgenden Abschnitten stelle ich noch Sperren und Signalobjekte vor. Die Tabelle in Abbildung 9.6 vergleicht das Semaphor mit diesen beiden. Auch Semaphor–Implementierungen nutzen oft andere Namen für die Operationen. In Swift heißen sie `signal` und `wait`,[3] was die Verwendung als Signal nahe legt. In Java sind es `acquire` und `release`,[4] was die Verwendung als Sperre oder zur Verwaltung betont. Ich habe oben bewusst die ursprünglichen Namen vorgestellt, um keine bestimmte Verwendung zu bevorzugen.

Operation	**Semaphor**	**Sperre**	**Signal**
Initialisierung	*n*	1	0
Synchron	`P`	`lock`	`wait`
Asynchron	`V`	`unlock`	`signal`

Abbildung 9.6.: Semaphor als Sperre oder Signal

Typische Erweiterungen des Semaphors erlauben das Hoch- und Runterzählen um mehr als 1. Als Alternative zum blockierenden `P` kann es eine versuchende Operation geben, die den Aufrufer niemals blockiert. Statt dessen meldet sie, ob das Runterzählen geklappt hat oder nicht. Versuchende Operationen stelle ich im folgenden Abschnitt 10.1 vor. In manchen Situationen kann es auch nützlich sein, alle wartenden Prozesse zu deblockieren, oder den Zähler auf 0 zu setzen, egal welchen Wert er zuvor hatte. Alle diese Erweiterungen bieten zum Beispiel die Semaphore in Java.
Ein Abfragen des Zählerstands ist möglich, hat aber rein informellen Charakter. Bis der Aufrufer etwas mit dem Wert anfängt, kann der Zählerstand sich durch andere Prozesse schon wieder geändert haben. Deshalb müssen alle Operationen, die vom Zählerstand abhängen, direkt im Semaphor implementiert sein und als atomare Aktionen ablaufen.

[3] `DispatchSemaphore`, abgerufen 2022-01-29
`https://developer.apple.com/documentation/dispatch/dispatchsemaphore`
[4] `java.util.concurrent.Semaphore`, abgerufen 2022-01-29
`https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Semaphore.html`

9.5. Knackpunkte

- Synchronisation bedeutet zeitliche Abstimmung zwischen Prozessen.
- Der Kern bietet Interaktionsobjekte an, über die sich Prozesse synchronisieren. Ein Beispiel ist das Semaphor, weitere folgen in Kapitel 10.
- Kritische Abschnitte in Befehlsströmen müssen atomar ablaufen.
- Exklusivsperren sind eine Möglichkeit, kritische Abschnitte zu schützen.
- Das Semaphor eignet sich als Exklusivsperre, zum Signalisieren und für weitere Synchronisationsprobleme. Siehe dazu das *Little Book of Semaphores*.[2]
- Bei Ablaufbeschreibungen gehe ich zur *Vereinfachung* davon aus, dass der Kern alle Methoden als kritische Abschnitte schützt. In der Praxis muss man sich selbst darum kümmern, dass das wirklich passiert. Und dass es nicht unnötig Parallelität einschränkt.

10. Ausgewählte Synchronisationsobjekte

Das Semaphor aus dem vorigen Kapitel 9 ist ein einfaches Synchronisationsobjekt. Es kombiniert einen Zähler mit einer Wartemenge und bietet zwei Operationen an, das synchrone `P` und das asynchrone `V`. In den folgenden Abschnitten stelle ich eine Auswahl anderer möglicher Kernobjekte zur Synchronisation vor. Auch sie bieten jeweils paarweise Operationen an. Zur synchronen und asynchronen Kopplung kommt noch die versuchende hinzu. Bei den Attributen erscheinen Flags sowie neue Arten von Mengen. Der Schwierigkeitsgrad der Beispiele steigt nach und nach. Ziel der Auswahl ist es, unterschiedliche Varianten im Verhalten aufzuzeigen.
Alle Beispiele in diesem Kapitel drehen sich um die Synchronisation, die zeitliche Abstimmung von Prozessen. Die Wettstein'sche Systemarchitektur nennt dies *Koordination*.[1] Ein anderer Aspekt der Interaktion ist das Übertragen von Daten zwischen Prozessen und Instanzen, die *Kommunikation*. Darauf gehe ich in Kapitel 11 ein.

Bei allen Interaktionsobjekten, nicht nur in den folgenden Beispielen, ist es wichtig, dass die Kernaktionen sämtlicher Operationen sich ergänzen und vervollständigen. Jede Änderung an einer Stelle muss an einer anderen Stelle rückgängig gemacht werden. Jede angefangene Kernoperation muss an einer anderen Stelle komplettiert, blockierte Prozesse deblockiert werden. Die Tabelle in Abbildung 10.1 zeigt, wie sich die die Kernaktionen am Semaphor ergänzen. Eine solche Tabelle kann man ebenso für andere Kernobjekttypen erstellen, auch wenn ein Objekt mehrere Methoden anbietet oder mehr als zwei Attribute verwendet. Das ist eine gute Kontrollmöglichkeit für Übungs- und Klausuraufgaben.

	Zähler	**Wartemenge**
`P`	runter	hinein, blockieren
`V`	hoch	heraus, deblockieren

Abbildung 10.1.: Kernaktionen ergänzen einander

10.1. Mutex

Mutex ist die Abkürzung für *mutual exclusion* (dt.: gegenseitiger Ausschluss), also die Funktion einer Exklusivsperre. Wie man das mit einem Binärsemaphor erreicht, habe ich in den beiden vorangegangenen Abschnitten erklärt. Abbildung 10.3 auf Seite 103 beschreibt ein Kernobjekt, das speziell für den Gebrauch als Mutex vorgesehen ist. Im Vergleich zum Semaphor aus Abbildung 9.3 auf Seite 94 genügt ein *Flag* (dt.: Merker) statt eines Zählers. Die Dateninvariante schließt wieder den ungültigen Fall aus, dass ein Prozess wartet, obwohl das Mutex frei ist. Die Methoden tragen sprechende Namen. Die Abläufe von `lock` und `unlock` entsprechen denen von `P` und `V`.
Als Erweiterung habe ich `lockTry` aufgenommen, ein *versuchendes* Sperren. Im Gegensatz zum synchronen `lock` blockiert es den Aufrufer auf keinen Fall. Entweder klappt das Setzen der Sperre oder nicht. Darauf muss der Aufrufer reagieren. Deshalb brauchen versuchende Operationen einen Rückgabewert, der den Erfolg oder Misserfolg anzeigt. Manche Implementierungen nutzen dazu den gleichen Mechanismus, mit dem sie auch Fehler zurückmelden. Allerdings ist Misserfolg kein Fehler, sondern einer von zwei möglichen, korrekten Abläufen der Operation.

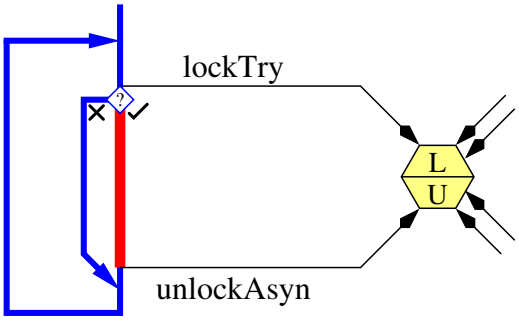


Abbildung 10.2.: Versuchendes Sperren

Abbildung 10.2 zeigt den Einsatz des versuchenden Sperrens als Interaktionsdiagramm. An der Aufrufstelle ist kein schwarzer Querbalken eingezeichnet, weil der Prozess nicht blockiert. Direkt nach dem Aufruf folgt eine Verzweigung. Im Erfolgsfall führt der Prozess einen kritischen Abschnitt aus und gibt die Sperre wieder frei. Bei Misserfolg springt der Prozess am kritischen Abschnitt vorbei und darf natürlich auch die Sperre nicht freigeben, da er sie nicht hält. Statt dessen könnte er zum Beispiel dem Benutzer eine Fehlermeldung anzeigen. Oder er leitet einen neuen Versuch zu einem späteren Zeitpunkt in die Wege. Deshalb habe ich den Befehlsstrom als Schleife gezeichnet. Wenn etwas immer wieder stattfinden soll, aber auch gelegentlich ausfallen darf, kann eine versuchende Operation das geeignete Mittel sein.
Das Mutex ist der einfachste, aber auch der fehlerträchtigste Weg zu gegenseitigem Ausschluss. Verlässt ein Prozess durch einen Fehler seinen kritischen Abschnitt, ohne die Sperre freizugeben, klappt nichts mehr. Sollen mehrere Funktionen mit kritischen Abschnitten einander aufrufen, muss man umständlich vermeiden, dass sie das Mutex sperren, wenn der Prozess es bereits hält, oder freigeben, wenn es von einer aufrufenden Funktion gesperrt wurde. Es gibt komfortablere Sperrmechanismen, zum Beispiel `synchronized` in Java, die solche Fälle berücksichtigen.

Mutex

Attribute:

Flag ob Sperre frei. Initial gesetzt, d.h. frei.

Wartemenge für Prozesse. Initial leer.

Dateninvariante:

Wenn **Flag** gesetzt, dann **Wartemenge** leer.

Methoden:

`lock` — *Sperren, synchron*

- **Flag** gesetzt, d.h. Sperre frei?

ja: **Flag** löschen, d.h. Sperre gesetzt

nein: Aufrufer blockieren und in **Wartemenge** stecken

`lockTry` — *Sperren, versuchend*

- **Flag** gesetzt, d.h. Sperre frei?

ja: **Flag** löschen und Erfolg zurückgeben

nein: Misserfolg zurückgeben

`unlock` — *Entsperren, asynchron*

- **Wartemenge** leer?

ja: **Flag** setzen, d.h. Sperre frei

nein: Prozess aus **Wartemenge** nehmen und deblockieren

Abbildung 10.3.: Attribute und Abläufe eines Mutex

10.2. Rendezvous

Das Rendezvous ist eine besondere Variante des Signalisierens. Klassisch kombiniert das Signalisieren eine asynchrone Operation `signal` mit einem synchronen `wait`. Sollen Signale gezählt und entsprechend oft abgeholt werden, nutzt man einen Zähler, wie beim Semaphor in Abschnitt 9.2. Repräsentiert das Signal eine Bedingung, die nur einfach gesetzt sein kann und beim ersten Abholen gelöscht wird, verwendet man statt dessen ein Flag. Dann ähnelt die Implementierung dem Mutex aus Abschnitt 10.1. Das Ausformulieren überlasse ich interessierten Lesern als Übungsaufgabe. Ein versuchendes `waitTry` kommt mit den gleichen Attributen aus.
Die Besonderheit am Rendezvous ist, dass beide beteiligten Prozesse synchron arbeiten: Wer zuerst da ist, wartet auf den anderen. Ein Semaphor genügt dafür nicht. Abbildung 10.4 beschreibt ein Rendezvous–Kernobjekt. Wie beim Signalisieren nenne ich die Operationen `signal` und `wait`. Allerdings funktionieren beide synchron, was mit im Namen steht. Da es zwei blockierende Operationen gibt, verwendet das Kernobjekt zwei Wartemengen. Die Dateninvariante schließt wieder den ungültigen Fall aus, dass in beiden Mengen ein Prozess wartet. Wenn ein Aufrufer von `signalSyn` und ein Aufrufer von `waitSyn` vorliegen, soll sofort ein Rendezvous stattfinden. Beide Abläufe entsprechen dem Schema für synchrone Operationen. Die Methoden verhalten sich exakt symmetrisch. Im Gegensatz zum normalen Signalisieren gibt es hier keine Richtung mehr, in der das Signal fließt. Deshalb sind auch die Namen der beiden Operationen nicht wirklich sinnvoll.
Auch das Rendezvous–Objekt kann man um versuchende Operationen ergänzen. Eine synchrone Operation auf der einen Seite führt mit einer synchronen oder versuchenden Operation auf der anderen Seite zum Rendezvous. Wenn allerdings beide Seiten nur versuchende Operationen aufrufen, kommt kein Rendezvous zustande. Statt selbst auf einen Partner zu warten, schauen beide nur nach, ob der andere schon wartet. Das kann nicht klappen. Es handelt sich dabei aber nicht um ein Problem des Kernobjekts. Es liegt in der Verantwortung der aufrufenden Prozesse, die Mechanismen des Kerns sinnvoll einzusetzen.

10.3. Signale mit Wertübergabe

Als Erweiterung der bisher vorgestellten Mechanismen soll nun beim Signalisieren ein Wert übermittelt werden. Nach der reinen Lehre [1] zählt das bereits zur Kommunikation, also zur Datenübertragung. In der Praxis schlägt man das Übertragen eines einzelnen Wertes, der in ein Register passt, noch der Synchronisation zu. So funktionieren die Echtzeit–Signale von POSIX. Ein weiteres Beispiel ist das Rendezvous in Plan 9, bei dem zwei Prozesse jeweils einen Wert an den Partner übergeben.[1] Mit der Datenübertragung beschäftigt sich Kapitel 11, wo die Daten nicht mehr in ein Register passen, sondern im Speicher liegen.

Anmerkung: *Der Begriff* Rendezvous *wird auch für einen Aufrufmechanismus zwischen Prozessen verwendet,[2], zum Beispiel in der Programmiersprache Ada.[3] Dabei fließen Nutz-*

[1] `http://man.cat-v.org/plan_9/2/rendezvous`
[2] `https://rosettacode.org/wiki/Rendezvous`
[3] `https://files.adacore.com/gnat-book/node22.htm`

Rendezvous

Attribute:

Wartemenge S für Signalgeber. Initial leer.

Wartemenge W für Signalnehmer. Initial leer.

Dateninvariante:

Mindestens eine **Wartemenge** muss leer sein.

Methoden:

`signalSyn` — *Signalisieren, synchron*

- **Wartemenge W** nicht leer?

ja: Prozess aus **Wartemenge W** nehmen und deblockieren

nein: Aufrufer blockieren und in **Wartemenge S** stecken

`waitSyn` — *Warten, synchron*

- **Wartemenge S** nicht leer?

ja: Prozess aus **Wartemenge S** nehmen und deblockieren

nein: Aufrufer blockieren und in **Wartemenge W** stecken

Abbildung 10.4.: Attribute und Abläufe eines Rendezvous

daten in Form von Parametern und Ergebnissen, in beide Richtungen. Das sprengt endgültig den Rahmen der Synchronisation, also der zeitlichen Abstimmung.

Abbildung 10.5 beschreibt ein Signalobjekt mit Wertübergabe. Die angebotenen Operationen sind `signalAsyn`, welches den zu übermittelnden Wert als Parameter erwartet, und `waitSyn`, das einen übermittelten Wert als Ergebnis liefert. Ein Zähler für Signale genügt jetzt nicht mehr. Statt dessen braucht man eine Menge, in der die signalisierten Werte liegen. Die Ähnlichkeit zum Rendezvous–Objekt aus Abbildung 10.4 ist unverkennbar. Dort liegen blockierte Prozesse in der Menge S, hier die beim Signalisieren übergebenen Werte. Die Abläufe der beiden Kernaktionen folgen dem bekannten Schema. Neu ist allerdings der Umgang mit dem Rückgabewert von `waitSyn`.
Findet die Kernaktion `waitSyn` einen Wert in der Menge S, so setzt sie diesen als Rückgabewert für den aufrufenden Prozess. Das ist kein `return`, wie man es aus vielen Programmiersprachen kennt. Die Kernaktion bricht an dieser Stelle nicht ab, sondern läuft zu Ende. Bei Bedarf könnte sie noch weitere Änderungen an den Attributen des Kernobjekts vornehmen. Erst am Ende der Kernaktion, wenn der aufrufende Prozess weiterrechnet, kommt der vorher gesetzte Rückgabewert zum Tragen.
Findet die Kernaktion `signalAsyn` einen Prozess in der Wartemenge W, so muss sie den vom Aufrufer übergebenen Wert als Rückgabewert dieses *anderen* Prozesses setzen. Das ist erst recht kein `return`, welches ja zum Aufrufer zurückkehren würde. Die Kernaktion schließt in diesem Fall gleich zwei Operationen ab, nämlich das Signalisieren des Aufrufers und das Warten des deblockierten Prozesses. Weil die Kernaktion zu Ende läuft, könnte sie ihre Änderungen genauso gut in anderer Reihenfolge vornehmen:

1. Prozess aus Wartemenge W nehmen
2. Rückgabewert des Prozesses setzen
3. Prozess deblockieren

Wichtig ist nur, dass am Ende einer Kernaktion alle notwendigen Änderungen erledigt sind. Wie in Abschnitt 9.3 erklärt bilden die Kernaktionen kritische Abschnitte, die durch geeignete Mechanismen im Kern atomar ablaufen. Außerhalb der Kernaktion gibt es den Zustand davor und den Zustand danach, aber keinen dazwischen. Auf die Übergabe von Parametern und Ergebnissen sowie den Umgang mit Registerinhalten gehe ich in Kapitel 16 genauer ein.

Signalobjekt mit Wertübergabe

Attribute:

Menge S für Signalwerte. Initial leer.

Wartemenge W für Prozesse. Initial leer.

Dateninvariante:

Mindestens eine der Mengen muss leer sein.

Methoden:

`signalAsyn(Wert)` — *Signalisieren mit Parameter*

- **Wartemenge W** nicht leer?

ja: Prozess aus **Wartemenge W** nehmen und deblockieren
Wert als Rückgabewert des Prozesses setzen

nein: **Wert** in **Menge S** legen

`waitSyn: Wert` — *Warten auf Rückgabewert*

- **Menge S** nicht leer?

ja: **Wert** aus **Menge S** nehmen und als Rückgabewert setzen

nein: Aufrufer blockieren und in **Wartemenge W** stecken

Abbildung 10.5.: Attribute und Abläufe eines Signals mit Werten

10.4. Flexible Kopplung

Als letzte Erweiterung in diesem Kapitel soll beim Signalisieren eines Wertes die Kopplungsform frei wählbar sein: asynchron wie in Abschnitt 10.3, synchron wie beim Rendezvous in Abschnitt 10.2, oder versuchend. Die Abbildungen 10.6 und 10.7 beschreiben ein solches Signalobjekt mit flexibler Kopplung. In der Menge für noch nicht abgeholte Signale muss es einerseits die Werte dieser Signale speichern, andererseits auch den blockierten Prozess für einen synchron signalisierten Wert. Dazu enthält diese Menge nun Tupel aus diesen beiden Komponenten. Bei asynchron signalisierten Werten gibt es keinen blockierten Prozess, also bleibt die zweite Komponente leer. Deshalb müssen die Operationen `waitSyn` und `waitTry` in Abbildung 10.7 beim Abholen eines Signals prüfen, ob sie einen Prozess deblockieren sollen oder nicht.
Das versuchende Signalisieren `signalTry` folgt dem Schema von `lockTry` beim Mutex in Abbildung 10.3. Im Gegensatz zu `signalAsyn` erfährt der signalisierende Prozess hier, ob ein anderer den Wert abholt. Diese Kopplungsform könnte man auch beim einfachen Signalobjekt mit Wertübergabe aus Abbildung 10.5 ergänzen, ohne dessen Attribute zu erweitern.
Die letzte aufgeführte Operation, `waitTry`, birgt eine kleine Herausforderung. Sie hat nämlich zwei Rückgabewerte: einen Erfolgsindikator und bei Erfolg den Signalwert. Je nach Programmiersprache muss man etwas tricksen, um eine Methode zu deklarieren, die so funktioniert. Bei den Ablaufbeschreibungen habe ich mich darum herum gemogelt, indem ich den Erfolgsindikator nicht in die Signaturen der versuchenden Methoden aufnehme.
Was zur freien Wahl der Kopplungsformen auf beiden Seiten noch fehlt, wäre ein asynchrones Warten. Schon die Bezeichnung ist ein Oxymoron. Die Wirkung dieser Operation wäre, einen Signalwert wegzuschmeißen. Zur Implementierung müsste man in der Wartemenge für Prozesse mehrfache Nulleinträge erlauben.

Eine freie Wahl zwischen asynchronem und synchronem Signalisieren am gleichen Objekt ist eher selten zu finden. Die Anwendungsfälle für die klassische Signalisierung und die Rendezvous–Signalisierung sind so unterschiedlich, dass man dafür auch einfachere, getrennte Mechanismen anbieten kann. Ich stelle die Kombination hier vor um aufzuzeigen, dass in den Wartemengen von Kernobjekten auch Zusatzinformationen liegen können. Das wird beim Thema Datenverschub in Kapitel 11 eine wichtige Rolle spielen.

Signalobjekt mit flexibler Kopplung (1/2)

Attribute:

Menge S für Signaltupel (Wert, Prozess). Initial leer.
Der Prozess darf `null` sein.

Wartemenge für Prozesse. Initial leer.

Dateninvariante:

Mindestens eine der Mengen muss leer sein.

Methoden:

`signalSyn(Wert)` — *Signalisieren synchron mit Parameter*

- **Wartemenge W** nicht leer?

ja: Prozess aus **Wartemenge W** nehmen und deblockieren
Wert als Rückgabewert des Prozesses setzen

nein: Aufrufer blockieren
(**Wert**, Aufrufer) in **Menge S** legen

`signalAsyn(Wert)` — *Signalisieren asynchron mit Parameter*

- **Wartemenge W** nicht leer?

ja: Prozess aus **Wartemenge W** nehmen und deblockieren
Wert als Rückgabewert des Prozesses setzen

nein: (**Wert**, `null`) in **Menge S** legen

Abbildung 10.6.: Attribute und Abläufe eines Signals mit flexibler Kopplung

Signalobjekt mit flexibler Kopplung (2/2)

Fortsetzung von Abbildung 10.6

`waitSyn: Wert` — *Warten auf Signalwert*

- **Menge S** nicht leer?

ja: Tupel (**Wert**, Prozess) aus **Menge S** nehmen
Wert als Rückgabewert setzen
Prozess deblockieren, falls nicht `null`

nein: Aufrufer blockieren und in **Wartemenge W** stecken

`signalTry(Wert)` — *Signalisieren versuchend mit Parameter*

- **Wartemenge W** nicht leer?

ja: Prozess aus **Wartemenge W** nehmen und deblockieren
Wert als Rückgabewert des Prozesses setzen
Erfolg an Aufrufer melden

nein: Misserfolg an Aufrufer melden

`waitTry: Wert` — *Signalwert holen, falls greifbar*

- **Menge S** nicht leer?

ja: Tupel (**Wert**, Prozess) aus **Menge S** nehmen
Wert als Rückgabewert setzen, Erfolg melden
Prozess deblockieren, falls nicht `null`

nein: Misserfolg melden

Abbildung 10.7.: Weitere Abläufe eines Signals mit flexibler Kopplung

10.5. Knackpunkte

- Ein Interaktionsobjekt bietet Operationen an, deren Aufrufe Interaktionsereignisse auslösen. Aufrufende Prozesse können ihren Ablauf an diese Ereignisse koppeln, zum Beispiel:

  Synchron: der Aufrufer wartet auf das Ereignis
  Asynchron: der Aufrufer wartet nicht, das Ereignis passiert irgendwann
  Versuchend: das Ereignis passiert gleich oder nie, der Aufrufer bekommt es mit

- Ein Interaktionsobjekt speichert Informationen, um Ereignisse zu erkennen und abzuarbeiten. Dazu gehören auch unvollständige Operationen. Typische Attribute sind: Flag, Zähler, Zeiger, Wartemenge, sonstige Menge.
  Für Zeiger siehe die Aufgabensammlung, kein Beispiel hier im Buch.

- Die Operationen und Methoden eines Interaktionsobjekts müssen sich ergänzen und vervollständigen. Jede Aktion braucht eine Gegenaktion. Zum Beispiel:
  - Flag setzen, löschen
  - Zähler erhöhen, vermindern; auf einen Wert setzen
  - Zeiger speichern, auf `null` setzen
  - (Warte-)Menge: Element einfügen, rausnehmen
  - Prozess blockieren, deblockieren

- Die Beispiele oben haben jeweils zwei komplementäre Operationen, mit verschiedenen Kopplungsformen. Interaktionsobjekte können mehr Operationen anbieten, zum Beispiel vier bei einer Leser–Schreiber–Sperre. Oder eine einzige, wie das Rendezvous in Plan 9, verlinkt in Abschnitt 10.3.

  In Klausuraufgaben sind der Phantasie dabei kaum Grenzen gesetzt. Aufgrund der Bearbeitungszeit läuft das bei mir auf drei bis vier Operationen hinaus.

- Bei Ablaufbeschreibungen gehe ich zur *Vereinfachung* davon aus, dass der Kern alle Methoden als kritische Abschnitte schützt.[4] In der Praxis muss man sich selbst darum kümmern, dass das wirklich passiert. Und dass es nicht unnötig Parallelität einschränkt.

[4] schon in Abschnitt 9.5 erwähnt, aber es ist wichtig

11. Datenverschub

Für die Interaktion zwischen Instanzen muss ein Betriebssystem auch eine Funktion anbieten, um Daten zu einer anderen Instanz zu schicken. Nicht nur in der Wettstein'schen Systemarchitektur [1] nennt man diesen Vorgang Kommunikation oder Datenübertragung. Allerdings ist Datenübertragung insgesamt ein sehr viel breiteres Thema. Deshalb habe ich für dieses Kapitel einen anderen, eher ungewöhnlichen Titel gewählt.
Hier geht es speziell um das Kopieren von kompakten, handhabbaren Speicherbereichen zwischen Adressräumen. Handhabbar bedeutet, dass eine Maximalgröße festgelegt ist. In COSY [3] lag diese bei 4500 Byte, so dass 4 KB Nutzdaten plus einige Zusatzinformationen in eine Nachricht passten. Allerdings sind die Speichergrößen seit dem letzten Jahrtausend deutlich gewachsen. Heute könnte auch eine Größe von 1 MB noch als handhabbar gelten. Ein guter Kompromiss läge vielleicht bei 64 KB.
Kapitel 10 stellt Mechanismen zur Synchronisation von Prozessen vor. Diese setze ich im Folgenden als bekannt voraus, insbesondere das Signalobjekt mit flexibler Kopplung und Wertübergabe aus Abschnitt 10.4 ab Seite 108. Statt des Wertes soll nun ein Speicherbereich übertragen werden. Abschnitt 11.1 bespricht die notwendigen Kopiervorgänge. Abschnitt 11.2 erläutert die Kopplung der beteiligten Prozesse an die Übertragung. Abschnitt 11.3 deckt auf, welche Vereinfachungen ich bei den Erklärungen gegenüber realen Systemen annehme. Abschnitt 11.4 fasst die wichtigsten Punkte dieses Kapitels zusammen.

11.1. Kopieren

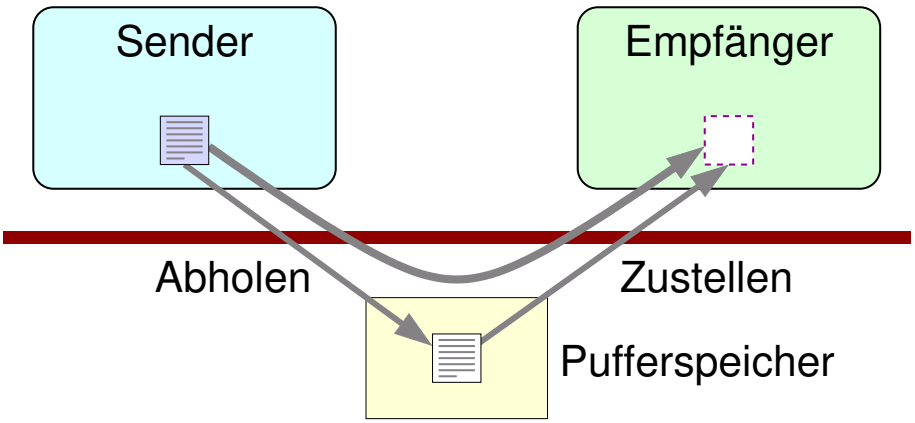


Abbildung 11.1.: Kopieren von Daten über Adressraumgrenzen

An jeder Übertragung sind zwei Prozesse beteiligt, Sender und Empfänger. Sie können in verschiedenen Adressräumen laufen. Da Prozesse nur auf Speicher in ihrem eigenen

Adressraum zugreifen dürfen, muss der Kern die Übertragung vornehmen. Der Sender übergibt Adresse und Größe der Nachricht an den Kern. Der Empfänger übergibt Adresse und Größe eines freien Speicherbereichs, in dem die Nachricht landen soll.
Ist der Empfänger bzw. das Ziel der Nachricht zum Zeitpunkt des Sendens bereits bekannt, dann kopiert der Kern sie direkt dorthin. Wird die Nachricht erst später zugestellt, gibt es zwei Möglichkeiten. Der Kern kann die Nachricht schon bei Senden abholen und in einen Pufferspeicher kopieren. Dann folgt beim Zustellen ein zweiter Kopiervorgang. Oder der Kern kann sich merken, wo die Nachricht liegt und sie später direkt von dort ans Ziel kopieren. Das spart einen Kopiervorgang. Dafür darf aber der Sender den Speicher der Nachricht nicht verändern oder freigeben, bevor sie zugestellt ist. Diese beiden Varianten heißen Wertablage und Referenzablage, in Anlehnung an die englischen Begriffe *call by value* und *call by reference* bei der Parameterübergabe von Funktionsaufrufen.
Die Entscheidung, ob Wert- oder Referenzablage angebracht ist, kann der Kern nicht alleine treffen. Doppeltes Kopieren bedeutet Zusatzaufwand, vor allem bei großen Datenmengen. Speicherbereiche beim Sender unbenutzbar zu machen, oder den Sender zu blockieren, kostet aber auch. Der Pufferspeicher im Kern ist beschränkt, so dass die Wertablage ohnehin an Grenzen stößt. Andererseits geht es um handhabbare Nachrichtengrößen. Wenn das Betriebssystem beide Varianten unterstützt, liegt es an den Entwicklerinnen der Instanzen und den Administratoren, sie sinnvoll einzusetzen und zu konfigurieren.
Beim Kopiervorgang panscht der Kern eine Übersetzungstabelle für die MMU so, dass er auf die Nachricht beim Sender bzw. den Zielbereich beim Empfänger zugreifen kann. Beim direkten Kopieren muss die Übersetzungstabelle also neben dem Adressraum des Kerns auch noch Teile beider Instanzenadressräume enthalten. Diese temporäre Übersetzungstabelle sollte nur auf dem Prozessor wirken, der gerade den Kopiervorgang durchführt.

11.2. Ereignisse

Es gibt verschiedene Verfahren, Sender und Empfänger zusammenzubringen. Ich stelle hier ein Kernobjekt Kanal vor, wie es in COSY zum Einsatz kam. Prozesse rufen Sende- und Empfangsoperationen auf, um einzelne Nachrichten zu übertragen. Es ist nicht notwendig, vorher eine Verbindung zu etablieren. Interaktionsereignisse entstehen, wenn eine Sende- auf eine Empfangsoperation trifft. Empfänger erhalten die nächste greifbare Nachricht. Der Kanal überträgt komplette Nachrichten, ohne sie aufzuteilen oder zu kombinieren. Abbildung 11.2 und Abbildung 11.3 beschreiben einen solchen Kanal. Aufgrund der vielen möglichen Sonderfälle und Fehlersituationen decken die Ablaufbeschreibungen nur grob die guten Fälle ab.
Wie das Signalobjekt mit flexibler Kopplung in Abbildung 10.6f ab Seite 109 benutzt der Kanal zwei Mengen, um anstehende Sende- oder Empfangsoperationen zu speichern. Mindestens eine der Mengen muss leer sein, da jedes Aufeinandertreffen von Sende- und Empfangsoperation zu einem Interaktionsereignis führt. Ein Interaktionsereignis markiert des Zustellen einer Nachricht. Unmittelbar nach dem Zustellen kann der Empfänger mit der Nachricht arbeiten, vorher nicht. Deshalb bietet der Kanal die Empfangsoperation nur mit synchroner und versuchender Kopplung an, aber nicht asynchron. Die Sendeoperation ist mit allen drei Kopplungsformen vorgesehen. Abläufe für die versuchenden Operationen fehlen. Mit den Vorkenntnissen aus Kapitel 10 sind sie leicht zu ergänzen.

Kanal mit Puffer (1/2)

Attribute:

Menge NP für Tupel (Nachrichtenreferenz, Prozess). Initial leer.
Der Prozess darf `null` sein.

Menge ZP für Tupel (Zielreferenz, Prozess). Initial leer.

Pufferspeicher begrenzter Größe. Initial leer.

Dateninvarianten:

Menge NP oder **Menge ZP** muss leer sein. Oder beide.

Wenn **Menge NP** leer, dann auch **Pufferspeicher** leer.

Methoden:

`sendAsyn` — *Senden, asynchron mit Wertablage*

- **Menge ZP** enthält Element?

ja: Tupel (Zielreferenz, Prozess) aus **Menge ZP** nehmen
Nachricht in Zielbereich kopieren
Prozess deblockieren

nein: **Pufferspeicher** hat Platz für Nachricht?
falls Nein: Sonderbehandlung, sonst weiter
Nachricht in **Pufferspeicher** kopieren
(Kopiereferenz, `null`) in **Menge NP** legen

Abbildung 11.2.: Attribute und Abläufe eines Kanals mit Puffer

Kanal mit Puffer (2/2)

Fortsetzung von Abbildung 11.2

`sendSyn` — *Senden, synchron mit Referenzablage*

- **Menge ZP** enthält Element?

ja: Tupel (Zielreferenz, Prozess) aus **Menge ZP** nehmen
Nachricht in Zielbereich kopieren
Prozess deblockieren

nein: Aufrufer blockieren
(Nachrichtenreferenz, Aufrufer) in **Menge NP** legen

`receiveSyn` — *Empfangen, synchron*

- **Menge NP** enthält Element?

ja: Tupel (Nachrichtenref., Prozess) aus **Menge NP** nehmen
Nachricht in Zielbereich kopieren
Prozess deblockieren, falls nicht `null`
Pufferspeicher freigeben, falls Nachricht von dort

nein: Aufrufer blockieren
(Zielreferenz, Aufrufer) in **Menge ZP** legen

`sendTry` — *Senden, versuchend*

`receiveTry` — *Empfangen, versuchen*

Abbildung 11.3.: Weitere Abläufe eines Kanals mit Puffer

Alle Sendeoperationen prüfen zunächst, ob ein Empfänger bekannt ist. Falls ja wird die Nachricht direkt zugestellt. In der Menge ZP liegen Tupel, die sowohl das Ziel der Nachricht beschreiben als auch den blockierten Empfangsprozess. Zum Ziel der Nachricht gehören der Adressraum, die Adresse und die Größe des Empfangspuffers. In der Praxis muss der Kanal prüfen, ob der Empfangspuffer groß genug ist, um die Nachricht aufzunehmen. Falls nicht könnte er die Sende- oder die Empfangsoperation mit einem Fehler abbrechen, oder auch beide. Oder er könnte nach einer anderen anstehenden Empfangsoperation suchen, die einen ausreichend großen Puffer benutzt. Letztlich könnte die Nachricht auch im Kanal bleiben, bis eine Empfangsoperation mit ausreichend großem Puffer stattfindet. Letzteres würde allerdings die erste Dateninvariante in Abbildung 10.6 verletzen. Im aufgeführten, einfachsten Fall kopiert die Sendeoperation die Nachricht ans Ziel und deblockiert den wartenden Empfänger.
Ist noch kein Empfänger bekannt, unterscheidet sich das Verhalten. Bei `sendAsyn` will der aufrufende Prozess weiterarbeiten und nicht auf das Zustellen der Nachricht warten. Hier ist es angebracht, die Nachricht in den Pufferspeicher des Kanals zu kopieren, damit der Sender den Speicherbereich der Nachricht sofort wieder nutzen kann. In der Menge NP landet eine Referenz auf die kopierte Nachricht, aber kein blockierter Prozess. Das setzt natürlich voraus, dass im Pufferspeicher des Kanals noch genug Platz für die Nachricht frei ist. Falls nicht, muss die Sendeoperation ein geeignetes *Überlaufverhalten* implementieren. Typische Reaktionsmöglichkeiten sind:

- Das Senden mit einem Fehler abbrechen.
- Alte Nachrichten löschen, bis genug Platz frei ist.
- Den Sender blockieren, bis genug Platz frei ist.

COSY implementierte die letzte Variante. Obwohl die Operation asynchron stattfinden soll, blockiert der Sender. Aber nicht bis zum Zustellen der Nachricht, sondern nur, bis durch Empfangsoperationen genug Platz im Puffer frei wird, um die Nachricht abzuholen. So entsteht automatisch eine Flusskontrolle, die den Zustrom neuer Nachrichten drosselt. Die ersten beiden Varianten sind einfacher zu implementieren, bringen aber eigene Schwierigkeiten mit sich. Abbrechen mit Fehler schiebt die Verantwortung für Flusskontrolle und neue Versuche in die sendende Instanz. Löschen alter Nachrichten macht den Kanal unzuverlässig, kann jedoch in manchen Szenarien ein passendes Vorgehen sein.
Bei `sendSyn` will der aufrufende Prozess warten, bis die Nachricht zugestellt ist. Hier bietet sich die Referenzablage an, da der Sender bis dahin nichts mit dem Speicher anfängt, in dem die Nachricht liegt. Die Nachrichtenreferenz umfasst in diesem Fall also den Adressraum des Senders sowie die Adresse und Größe der Nachricht. Zusätzlich steht der blockierte Prozess im Tupel, das in der Menge NP landet.
Das synchrone Empfangen `receiveSyn` implementiert die Gegenstücke zu den Aktionen der Sendeoperationen. Es schaut zunächst nach, ob in der Menge NP bereits eine Nachricht vorliegt. Falls ja kopiert es sie ans Ziel. Auch hier muss eine echte Implementierung prüfen, dass der Empfangspuffer groß genug ist. Außerdem wären zusätzliche Informationen über die Nachricht interessant, zum Beispiel deren Größe. In COSY galt eine Konvention, dass am Anfang jeder Nachricht deren Größe und ggfs. eine Kanalkennung für die Antwort steht. Ein solches Vertrauen in den Sender führt aber in der Praxis zu katastrophalen

Sicherheitslücken.[1] Nach dem Zustellen der Nachricht deblockiert die Empfangsoperation den Sender, falls dieser wartet. Außerdem gibt sie den Pufferspeicher im Kanal frei, falls die Nachricht dort lag. Wenn keine Nachricht vorliegt, blockiert der Empfänger und landet mit der Referenz auf den Empfangspuffer in der Menge ZP.

Eine gängige Erweiterung auch einfacher Kanalkonzepte ist das selektive Empfangen. Jeder Sender gibt seiner Nachricht einen Typ mit, letztlich eine Zahl. Empfänger können dann gezielt Nachrichten eines bestimmten Typs abholen. Mit dieser Erweiterung dürfen in beiden Mengen gleichzeitig Elemente liegen. So funktionieren zum Beispiel die Operationen `msgsnd` und `msgrcv` in POSIX.[2]

11.3. Zeitaufwand

Bei den Ablaufbeschreibungen in Abschnitt 11.2 stehen die Kopiervorgänge in den Kernaktionen. In Abschnitt 9.3 behaupte ich, dass alle Kernaktionen kritische Abschnitte sind, die der Kern nur sequentiell ausführt. Und laut Abschnitt 12.2 laufen Kernaktionen als Unterbrechungsbehandlungen ab. All dies sind *Vereinfachungen*, um ein grobes Verständnis des Kerns aus Entwicklersicht zu vermitteln.
Das Umkopieren von Kilobyte an Daten hat in einer Unterbrechungsbehandlung oder einem kritischen Abschnitt *nichts verloren*. Beides muss schnell ablaufen, damit bald weitere Unterbrechungen bearbeitet werden, andere Befehlsströme ihre kritischen Abschnitte ausführen können. In COSY waren solche Vereinfachungen akzeptabel, weil es keine Peripherie gab, auf jedem Rechenknoten nur ein Prozessor arbeitete und Programme ohnehin im Batch–Modus abliefen, also nicht schnell reagieren mussten. Außerdem war die PowerPC–Portierung [3] nur eine Diplomarbeit.
In einem praxistauglichen Betriebssystem müssen länger laufende Vorgänge wie das Umkopieren von Daten in langlebigen Befehlsströmen stattfinden, die unterbrochen und später fortgesetzt werden können. Dazu gibt es verschiedene Ansätze, die ich in Abschnitt 16.3 schon angesprochen habe. Eine Möglichkeit ist, die Prozesse von Instanzen bei Kernaufrufen in einen Kern–Modus umzuschalten. Damit ruht der Befehlsstrom der Instanz, aber der Prozess bekommt weiter Rechenzeit, um Code des Kerns auszuführen. Dort kann er Daten umkopieren, nachdem er in einem kritischen Abschnitt Pufferspeicher reserviert oder den Zielpuffer ermittelt hat. Eine andere Möglichkeit ist, reine Kernprozesse aufzusetzen, die Aufträge zum Umkopieren von Daten abarbeiten. Dann muss der Sender während des Abholens oder der Empfänger während des Zustellens einer Nachricht blockiert werden, bis das Umkopieren erledigt ist.
Wer an der Entwicklung eines Betriebssystemkerns mitarbeitet, muss sich mit solchen Details beschäftigen. Im Rahmen meiner Vorlesung betrachte ich sie jedoch als Ballast, der den allermeisten Studierenden nicht weiterhilft und das Verständnis der wesentlichen Zusammenhänge unnötig erschwert.

[1] `https://xkcd.com/1354/` *Heartbleed*–Bug in OpenSSL
[2] `https://linux.die.net/man/2/msgsnd` Manpage zu `msgsnd` und `msgrcv`

11.4. Knackpunkte

- Kommunikation erfordert Datenübertragung. Für gewöhnlich durch Kopieren.
- Kopieren über Adressraumgrenzen ist eine Aufgabe für den Kern.
- Das Zustellen von Daten beim Empfänger ist ein Interaktionsereignis.
- Das Abholen von Daten beim Sender ist ein anderes Interaktionsereignis.
- Kanäle sind eine Möglichkeit, Sender und Empfänger zusammenzubringen.
- Nachrichten sind eine Möglichkeit, Daten für den Transport zusammenzufassen.
- Sender und Empfänger übergeben dem Kanal Referenzen auf Speicherbereiche.
- Ist noch kein Empfänger da, kann der Kanal die Referenz oder den Inhalt (Wert) einer gesendeten Nachricht ablegen.
- Referenzablage spart einen Kopiervorgang. Wertablage erlaubt dem Sender, seinen Speicher wieder zu verwenden. Was besser ist, muss der Sender wissen und passende Operationen aufrufen. Die Kopplungsform des Senders spielt dabei eine Rolle.
- Es gibt auch andere Möglichkeiten, Kommunikation zu strukturieren.

12. Umschalten

Die Prozesse aus Kapitel 8 repräsentieren Befehlsströme, zwischen denen das Betriebssystem umschaltet. Im Folgenden betrachten wir dieses Umschalten etwas genauer. Abschnitt 12.1 führt eine neue Art von Befehlsströmen ein, nämlich kurze. Abschnitt 12.2 beschreibt Kernaktionen, die als kurze Befehlsströme ablaufen. Sie können zwischen Prozessen, also langlebigen Befehlsströmen, umschalten. Abschnitt 12.3 erklärt, wie zwei oder mehr Kernaktionen zu verschiedenen Zeitpunkten eine blockierende Kernoperation implementieren. Abschnitt 12.4 fasst die wichtigsten Punkte dieses Kapitels zusammen. Mehr Details zu Kernaktionen und Umschaltvorgängen liefert Kapitel 16.

12.1. Befehlsströme

Ein Prozess ist ein Befehlsstrom, der für gewöhnlich stückchenweise abläuft. Mal erhält er einen Prozessor und kommt voran, mal ist kein Prozessor für ihn frei, mal ruft er eine blockierende Operation auf und muss warten. Das erzeugt Verwaltungsaufwand für Kernobjekte, Bereit- und Wartemengen, wie in Kapitel 8 beschrieben. Weil Prozesse langlebige Befehlsströme darstellen, bleibt dieser Aufwand in vertretbaren Grenzen. Prozentual zur Lebensdauer oder genutzten Rechenzeit eines Prozesses ist die Rechenzeit für seine Zustandswechsel gering.
In der Infrastruktur gibt es zudem kurze Befehlsströme, speziell zur Unterbrechungsbehandlung. Sie beginnen spontan, als Reaktion auf einen Vorfall im System, und laufen direkt zu Ende. Eine aufwendige Verwaltung wäre undenkbar. Das schränkt diese Befehlsströme aber ein. Insbesondere können sie keine blockierenden Operationen durchführen. Außerdem sollten sie möglichst schnell zum Ende kommen, damit der Prozessor auf den nächsten Vorfall reagieren kann.

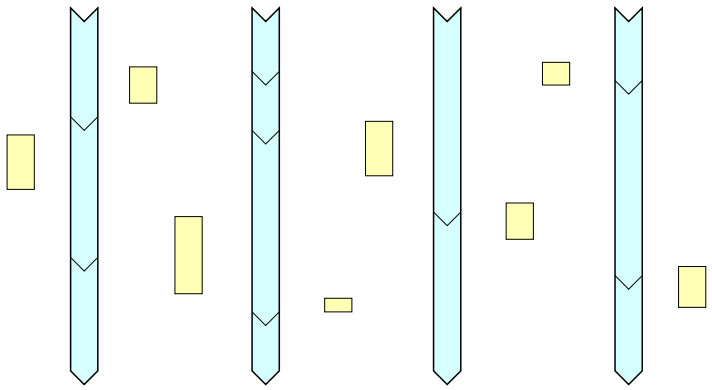


Abbildung 12.1.: Langlebige und kurze Befehlsströme

Abbildung 12.1 zeigt langlebige Befehlsströme als blaue Streifen, die in Abschnitten von oben nach unten verlaufen. Gelbe Kästchen stehen für kurze Befehlsströme, die am Stück ablaufen. Der Vergleich mit Abbildung 8.1 auf Seite 77 legt nahe, dass kurze Befehlsströme das Umschalten zwischen langlebigen implementieren. Jeder Prozessor in einem Rechner kann zu jedem Zeitpunkt nur einen Befehlsstrom ausführen. Das Umschalten von einem Prozess zu einem anderen erfolgt in zwei Schritten: Umschalten von einem langlebigen zu einem kurzen Befehlsstrom und weiter zum nächsten langlebigen.
Als vereinfachende Annahme gehe ich davon aus, dass Instanzen nur mit Prozessen arbeiten, während die Infrastruktur ausschließlich kurze Befehlsströme verwendet. Ersteres stimmt, Letzteres im Allgemeinen nicht. Aus Sicht einer Entwicklerin, deren Code in Instanzen abläuft, spielt es aber keine Rolle, wie sich der Kern Rechenzeit beschafft. Einen genaueren Blick auf die Vorgänge im Kern wirft Kapitel 16.

Anmerkung: *Instanzen können ebenfalls kurze Befehlsströme verwenden, zum Beispiel mit dem async/await–Paradigma. Auch Unterbrechungsbehandlung ist möglich. Die Infrastruktur vergibt Rechenzeit aber trotzdem nur an Prozesse, welche dann innerhalb der Instanz verschiedene kurze Befehlsströme aufgreifen.*

12.2. Kernaktionen

Als Kernaktion bezeichne ich die Ausführung eines kurzen Befehlsstroms, typischerweise als Reaktion auf eine Unterbrechung. Abbildung 12.2 zeigt eine Kernaktion ohne Prozessumschaltung, also den einfachsten Fall. Der Befehlsstrom des Prozesses wird unterbrochen, der Kern führt eine Aktion aus und springt dann zum gleichen Prozess zurück.

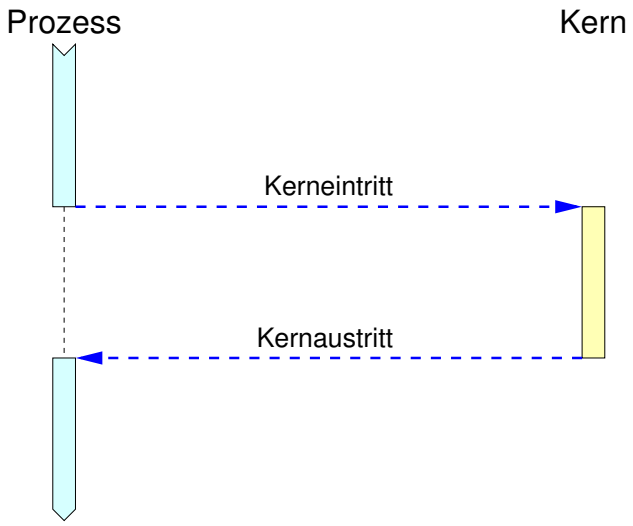


Abbildung 12.2.: Kernaktion ohne Umschalten

Diese und die folgenden Abbildungen von Befehlsströmen verwenden eine einheitliche Zeitachse, vertikal von oben nach unten. Der Wechsel eines Prozessors von einem Befehlsstrom zu einem anderen ist als horizontaler, gestrichelter Pfeil dargestellt. Kerneintritte

führen von einem Prozess zum Kern, Kernaustritte vom Kern zu einem Prozess. Diese Darstellung ist detaillierter als Abbildung 8.1 auf Seite 77, wo jeder Befehlsstrom seine eigene Zeitachse hat und der Kern gar nicht auftaucht.
Abbildung 12.3 zeigt das Umschalten von einem Prozess zu einem anderen in einer Kernaktion. Zunächst rechnet P1, dann erfolgt ein Kerneintritt. Im Verlauf der Aktion entscheidet der Kern, zu einem anderen Prozess umzuschalten. Vielleicht blockiert P1. Oder seine Zeitscheibe ist abgelaufen und ein anderer Prozess steht bereit, um den Prozessor zu übernehmen. Oder die Kernaktion hat einen anderen Prozess deblockiert, dessen Priorität höher ist als die von P1.
Jedenfalls fällt in der Kernaktion die Entscheidung zum Umschalten. Deren Auswirkung zeigt sich wenig später, beim Kernaustritt. Statt zurück zum Befehlsstrom von P1 springt der Prozessor zu dem von P2. Die Zustandsübergänge beim Umschalten, nämlich das Zurückstellen von P1 und das Aufgreifen von P2, sind nur Änderungen an internen Datenstrukturen des Kerns. Diese Datenstrukturen bestimmen aber, zu welchem Prozess der folgende Kernaustritt führt.

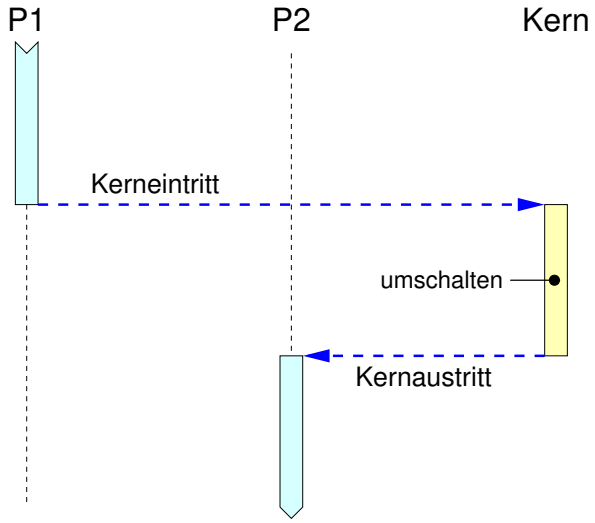


Abbildung 12.3.: Kernaktion mit Umschalten

12.3. Kernoperationen

Systemfunktionen der Infrastruktur übernehmen viele Aufgaben, die sich nicht mit einem einzigen, kurzen Befehlsstrom erledigen lassen. Zum Beispiel bieten Betriebssysteme eine Operation `sleep` an, die den Aufrufer für längere Zeit blockiert. Zur Unterscheidung von Aufgabe und Durchführung verwende ich die Begriffe Kernoperation bzw. Kernaktion. Instanzen rufen Kern*operationen* auf, welche längere Zeit in Anspruch nehmen. Kern*aktionen* sind kurze Befehlsströme, die zu verschiedenen Zeitpunkten stattfinden und jeweils einen Teil der Operation erledigen. Im Fall von `sleep` führt der Aufruf zu einer Kernaktion, die den aufrufenden Prozess blockiert. Nach Ablauf der Wartezeit folgt eine weitere Kernaktion, die den Prozess wieder deblockiert und die Operation abschließt. Die erste Kernaktion muss die zweite vorbereiten.

In manchen Fällen genügt eine einzige Kernaktion für die gesamte Operation, etwa wenn die gewünschte Wartezeit 0 ist. In anderen Beispielen können mehr als zwei Kernaktionen notwendig sein. Aufgerufene Kernoperationen können Ergebnisse an den aufrufenden Prozess zurückgeben. Der Kern führt aber auch eigenständig Operationen und Aktionen aus, die kein Prozess aufruft. Beispiele dafür sind die Reaktion auf Seitenfehler der MMU, wie in Abschnitt 14.3 beschrieben, oder das Umschalten nach Ablauf einer Zeitscheibe.
Abbildung 12.4 zeigt eine Kernoperation, die zwei Kernaktionen benötigt. Zum Beispiel wenn ein Prozess eine Sperre setzen will, die gerade nicht frei ist. Die Kernaktion beim Aufruf erkennt, dass eine notwendige Bedingung nicht erfüllt ist und blockiert den Prozess, wie in Abschnitt 8.3 besprochen. In der Abbildung ist dieser Zeitpunkt mit einem roten Balken links markiert. Beim Kernaustritt springt der Prozessor zu einem der aktiven Prozesse. In der Folge können viele weitere Kernaktionen stattfinden, die für die Kernoperation des blockierten Prozesses keine Rolle spielen.

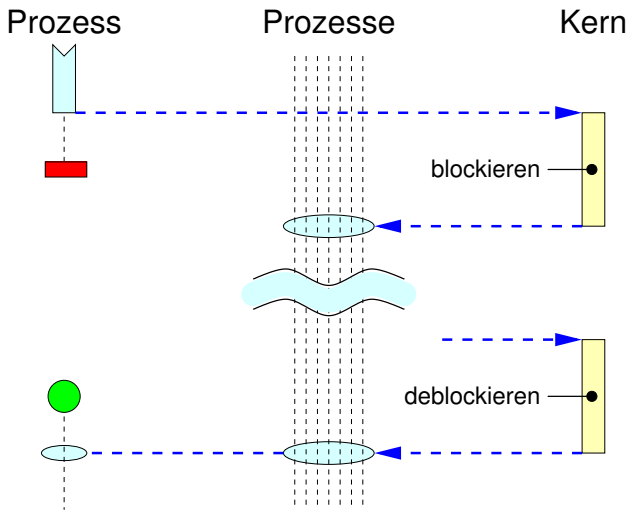


Abbildung 12.4.: Kernoperation mit Blockieren und Deblockieren

Irgendwann findet eine Kernaktion statt, die die Bedingung erfüllt. Zum Beispiel gibt ein anderer Prozess die Sperre frei, und der blockierte Prozess erhält sie. Diese Kernaktion schließt gleich zwei Operationen ab, nämlich die des freigebenden und die des blockierten Prozesses. Sie deblockiert deshalb den Prozess, der die Sperre erhält. Dieser Zeitpunkt ist mit einem grünen Kreis links markiert. Ob der Kern beim Deblockieren auch zu diesem Prozess umschaltet, hängt von der konkreten Situation ab, zum Beispiel von den Prioritäten der beteiligten Prozesse. Jedenfalls springt der Kernaustritt nach Abschluss der zweiten dargestellten Kernaktion zu einem der aktiven Prozesse, zu denen der gerade deblockierte auch wieder zählt.
Die meisten blockierenden Kernoperationen ruft ein Prozess explizit auf. Zum Beispiel um gewisse Zeit zu schlafen, eine Nachricht zu empfangen oder ähnliches. Kapitel 10 präsentiert eine Auswahl von Kernobjekten mit blockierenden Operationen. Das Blockieren ohne expliziten Aufruf ist typisch für einen Seitenfehler bei der virtuellen Speicherverwaltung. Mehr Informationen dazu stehen in Abschnitt 19.2.

12.4. Knackpunkte

- Unterbrechungen lösen Kern*aktionen* aus.
- Kernaktionen sind kurze Befehlsströme, die zwischen der Ausführung von Prozessen eingestreut werden.
- Kernaktionen laufen direkt zu Ende. Sie dürfen nicht blockieren.
- Kernaktionen können von einem Prozess zu einem anderen umschalten.

- Prozesse rufen Kern*operationen* auf.
- Kernoperationen können Ergebnisse zurückgeben.
- Kernoperationen erfordern eine oder mehrere Kernaktionen. Zu verschiedenen Zeitpunkten, falls mehrere.
- Eine Kernaktion kann den aufrufenden bzw. unterbrochenen Prozess blockieren.
- Dann muss eine spätere Kernaktion den Prozess wieder deblockieren, um die Kernoperation abzuschließen.
- Eine Kernaktion kann einen oder mehrere Prozesse deblockieren. Damit kann sie auch Teil mehrerer Kernoperationen sein.

13. Adressen

Aus Kapitel 3 ist bekannt, dass ein Programm im Hauptspeicher aus einer Anzahl von Segmenten besteht, die in einen Adressraum eingeblendet sind. Der Zugriff auf den Hauptspeicher erfolgt über Adressen. In erster Näherung stellen wir uns vor, dass alle Bytes des Hauptspeichers durchnummeriert sind. Eine Adresse ist die Nummer eines Bytes. Jedes Segment liegt kompakt an aufeinanderfolgenden Adressen. Abbildung 13.1 zeigt eine mögliche Lage der Segmente eines Programms. Es handelt sich um die gleichen Segmente wie in Abbildung 3.1 auf Seite 34, mit nur einem einzigen Stapel.

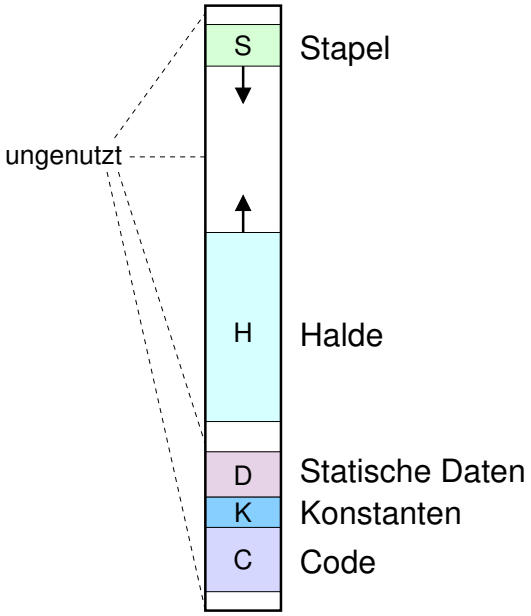


Abbildung 13.1.: Segmente im Adressraum

Der gesamte Balken repräsentiert einen Adressraum. Auf einem 32–Bit–Rechner reicht er von Adresse 0 unten bis Adresse $2^{32} - 1$ oben. Segmente dürfen sich nicht überlappen. Jede Speicherstelle enthält nur einen Byte–Wert. Das kann zum Beispiel ein Wert des Code–Segments oder des Konstanten–Segments sein, aber nicht beides gleichzeitig. Lücken zwischen Segmenten sind erlaubt, wie die weißen Bereiche zeigen. Auf diese Adressen darf das Programm nicht zugreifen.
Code, Konstanten und statische Daten haben eine feste Größe. Deshalb können sie direkt hintereinander liegen. Stapel und Halde wachsen nach Bedarf. Traditionell liegt der Stapel oben und wächst nach unten, während die Halde unten liegt und nach oben wächst. Sollten beide Segmente so groß werden, dass sie sich überlappen, kommt es zur Katastrophe. Im besten Fall bricht das Programm wegen Speichermangels ab, im schlimmsten Fall läuft es weiter und produziert falsche Ergebnisse.

Abbildung 13.1 ist nicht maßstabsgetreu. Ein 32–Bit–Rechner kann 4 Gigabyte adressieren. Segmente sind in den meisten Fällen sehr viel kleiner, im Bereich von Megabyte oder weniger. Das macht es möglich, viele Programme gleichzeitig im Speicher zu halten. In Systemen mit 64–Bit–Adressen umfasst der verfügbare Hauptspeicher nur einen kleinen Bruchteil des Adressraums, die Segmente wiederum nur einen Bruchteil des verfügbaren Hauptspeichers. Die Größenverhältnisse in Abbildung 13.1 passen eher zu 16–Bit–Adressen mit 64 Kilobyte Hauptspeicher. In diesen Rechnern aus den 80ern lief für gewöhnlich nur ein Programm.
Abschnitt 13.1 zeigt die Schwierigkeiten auf, wenn mehrere Programme laufen und alle die physischen Adressen des Hauptspeichers verwenden. Abschnitt 13.2 stellt logische Adressen vor, welche nur innerhalb einer Instanz gelten. Abschnitt 13.3 erklärt, auf welche Weise sogenannte MMUs (engl.: *Memory Management Units*) logische in physische Adressen übersetzen. Abschnitt 13.4 fasst die wichtigsten Punkte dieses Kapitels zusammen.

13.1. Physische Segmente

Abbildung 13.2 zeigt den schematischen Aufbau eines Rechners. Ein Systembus, also eine Reihe von elektrischen Leitungen, verbindet Prozessoren (CPU), Hauptspeicher (RAM) und weitere Komponenten (Peripherie). Die elektrischen Leitungen übertragen Adressen, Daten und mehr. Moderne Rechner sind tatsächlich weitaus komplizierter aufgebaut. Das Programmiermodell für Zugriffe beruht aber nach wie vor auf dieser einfachen Sichtweise.

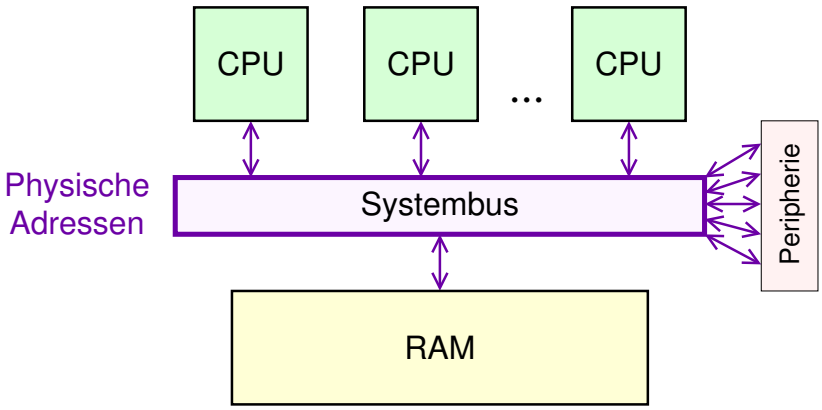


Abbildung 13.2.: Komponenten am Systembus

Der Hauptspeicher ist eine passive Komponente. Er liest jede Adresse auf dem Systembus und reagiert auf diejenigen, für die er zuständig ist. Bei einem Lesezugriff liefert er die Daten an der Adresse, bei einem Schreibzugriff speichert er Daten dort ab. Prozessoren sind aktive Komponenten. Sie legen Adressen auf den Systembus, um Lese- und Schreibzugriffe durchzuführen. Zur Peripherie gehören sowohl aktive als auch passive Komponenten.
Adressen auf dem Systembus bezeichne ich als physische Adressen. Ihre Bedeutung wird dadurch festgelegt, welche Komponente darauf reagiert. Um auf eine bestimmte Stelle des Hauptspeichers zuzugreifen muss genau deren Adresse auf dem Systembus liegen. Dabei spielt es keine Rolle, welcher Prozessor den Zugriff durchführt. Ebensowenig spielt es eine Rolle, welches Programm der zugreifende Prozessor gerade ausführt.

Abbildung 13.3 zeigt die Speicherbelegung, wenn mehrere Programme gestartet werden. Bei der ersten Instanz links ist der Speicher noch frei. Die Segmente der Instanz liegen an aufeinanderfolgenden, physischen Adressen. Die zweite Instanz in der Mitte muß andere physische Adressen verwenden. Der Programmlader oder der Kern legt diese Segmente also in die freien Lücken. Auch jede weitere Instanz, die gestartet wird, muss in die dann noch freien Lücken passen. Falls Instanzen das gleiche Programm ausführen oder die gleiche dynamische Bibliothek laden, können sie Segmente für Code und Konstanten gemeinsam verwenden.

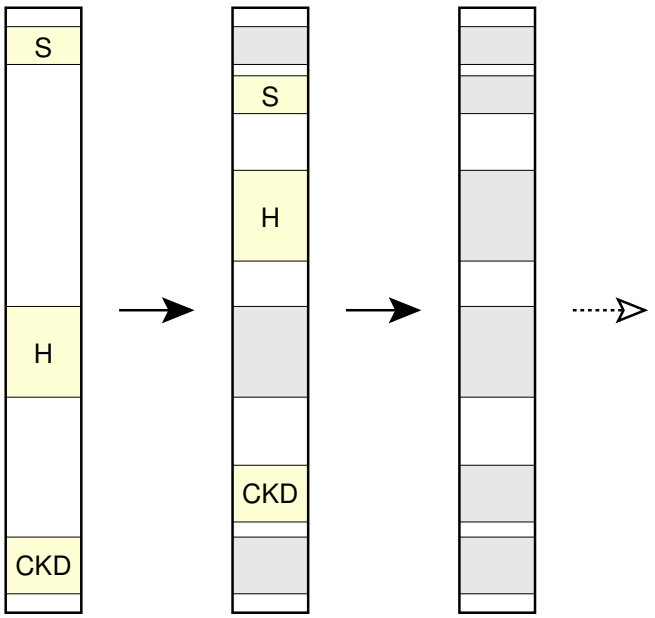


Abbildung 13.3.: Mehrere Instanzen mit physischen Segmenten

Der Speicher fragmentiert im Allgemeinen nicht ganz so schnell, wie man nach Abbildung 13.3 glauben könnte. Die Segmente sind, wie oben erwähnt, im Vergleich zum Hauptspeicher deutlich kleiner. Nichtsdestotrotz kann es vorkommen, dass die Anforderung eines großen, zusammenhängenden Speicherbereichs fehlschlägt, weil es in keine Lücke mehr passt. Betriebssysteme aus dieser Zeit implementierten aufwendige Maßnahmen, die das Problem aber nicht lösten und selbst fehleranfällig waren.[1]
Ein weiteres Problem ist der fehlende Schutz vor ungültigen Speicherzugriffen. Jede Instanz kann durch Fehler oder böse Absicht auf Adressen zugreifen, die nicht zu ihren Segmenten gehören. Sie kann dabei nicht nur fremden Speicher auslesen, sondern auch überschreiben. Code und Konstanten sind ebenfalls nicht vor Veränderungen geschützt. Verbotene Schreibzugriffe gefährden neben anderen Instanzen auch den Kern. Abhilfe schafft erst der Einsatz von MMUs.

[1] `https://en.wikipedia.org/wiki/Classic_Mac_OS_memory_management` abgerufen 2025–01–22

13.2. Logische Segmente

Eine MMU sitzt zwischen einem Prozessor und dem Systembus, wie in Abbildung 13.4 dargestellt. Sie übersetzt die logischen Adressen des Befehlsstroms auf dem Prozessor in physische Adressen auf dem Systembus. Indem der Kern für jede Instanz eine andere Übersetzung definiert, beschränkt er die Speicherzugriffe der Befehlsströme auf die Segmente ihrer Instanz. Der Adressraum einer Instanz wird damit von einer theoretischen Idee zu einem technisch durchgesetzten Konzept. MMUs machen aus Adressräumen Gefängnisse für Instanzen und deren Befehlsströme.

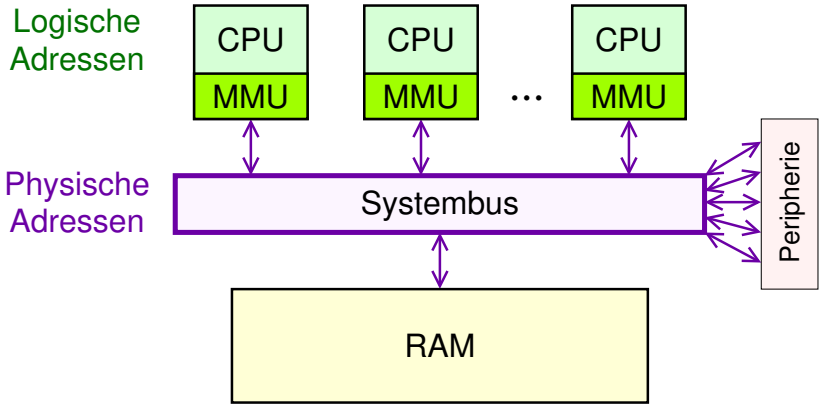


Abbildung 13.4.: MMUs am Systembus

Abbildung 13.5 zeigt die Adressräume von drei Instanzen bei Verwendung logischer Adressen. Segmente im gleichen Adressraum dürfen sich nach wie vor nicht überlappen. Aber die Segmente in verschiedenen Adressräumen sind voneinander entkoppelt. Zum Beispiel können die Halden an überlappenden logischen Adressen liegen, wie es die Zeichnung zeigt. Durch eine unterschiedliche Übersetzung pro Adressraum liegen die Speicherinhalte trotzdem an jeweils anderen physischen Adressen. Sollen Instanzen Code und Konstanten gemeinsam verwenden, definiert der Kern die Übersetzungen so, dass diese Segmente bei den beteiligten Instanzen an den gleichen physischen Adressen landen.
Für die Übersetzung teilt man sowohl die logischen Segmente als auch den physisch adressierten Hauptspeicher in *Seiten* fester Größe ein. Typische Seitengrößen sind 4 Kilobyte oder 64 Kilobyte. Segmente müssen ein exaktes Vielfaches der Seitengröße umfassen. Eine Übersetzungstabelle definiert für jede logische Seite, in welcher physischen Seite deren Inhalte liegen. Aufeinanderfolgende logische Seiten liegen im RAM verstreut, je nachdem wo bei der Anforderung gerade eine physische Seite frei war. Abbildung 13.6 auf Seite 134 veranschaulicht diesen Zusammenhang.
Der folgende Abschnitt 13.2.1 geht auf Schutzaspekte der Adressübersetzung sowie auf Vorteile bei der Speicherverwaltung ein. Eine Beschreibung der Übersetzungstabellen enthält Abschnitt 13.3. Die Verwaltung der Tabellen ist Thema von Kapitel 14, fortgeschrittene Einsatzmöglichkeiten zeigt Kapitel 19 auf.

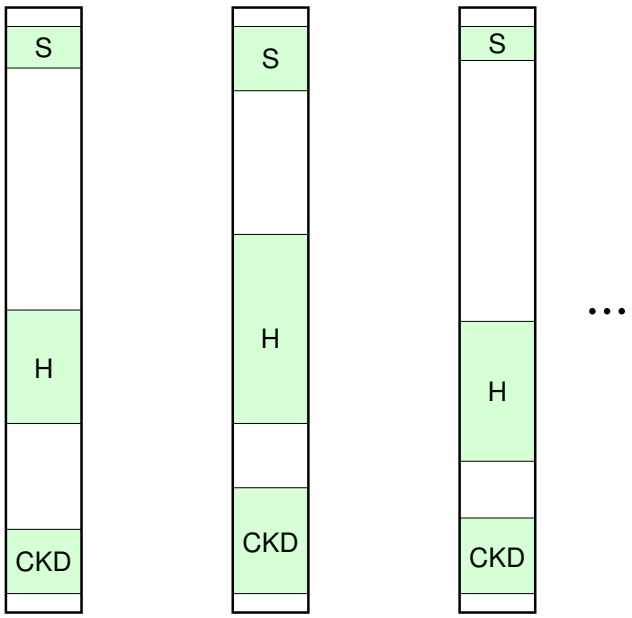


Abbildung 13.5.: Mehrere Instanzen mit logischen Segmenten

13.2.1. Speicherschutz

Für logische Seiten, die nicht zu einem Segment gehören, ist keine Übersetzung definiert. Die MMU löst beim Zugriff auf eine solche Seite eine Unterbrechung aus, einen *Seitenfehler* (engl.: *page fault*). Auch für definierte Übersetzungen können Einschränkungen gelten, zum Beispiel dass die Seite nur gelesen, aber nicht geschrieben werden darf. So schützt die MMU Code und Konstanten vor Veränderungen.
Eine Seite ist klein im Vergleich zum Adressraum oder zum Hauptspeicher. Aber sie ist groß im Vergleich zu den meisten Objekten auf einer Halde, oder zu Stack Frames auf einem Stapel. Die Adressübersetzung durch MMUs kann also nicht verhindern, dass eine Instanz auf einen Teil der Halde oder des Stapels zugreift, wo gerade keine gültigen Daten liegen. Sie sorgt aber dafür, dass andere Instanzen auf der Halde oder dem Stapel dieser Instanz keinen Unfug treiben. Der Speicherschutz durch MMUs wirkt auf Ebene der Adressräume und Segmente, nicht innerhalb eines Segments.
Das Zusammensetzen logischer Segmente aus verstreuten physischen Seiten erleichtert die Speicherverwaltung des Betriebssystems enorm. Statt wie in Abbildung 13.3 nach zusammenhängenden, freien Bereichen im Hauptspeicher zu suchen, liefert es einfach die passende Anzahl aus einer Liste freier Seiten. Innerhalb einer Instanz ändert sich nichts daran, wie diese Stapel und Halde in kleineren Stücken verwaltet. Die Speicherverwaltung einer Instanz profitiert allenfalls davon, dass sie Halde oder Stapel zur Laufzeit vergrößern kann, ohne dass andere Instanzen in die Quere kommen.

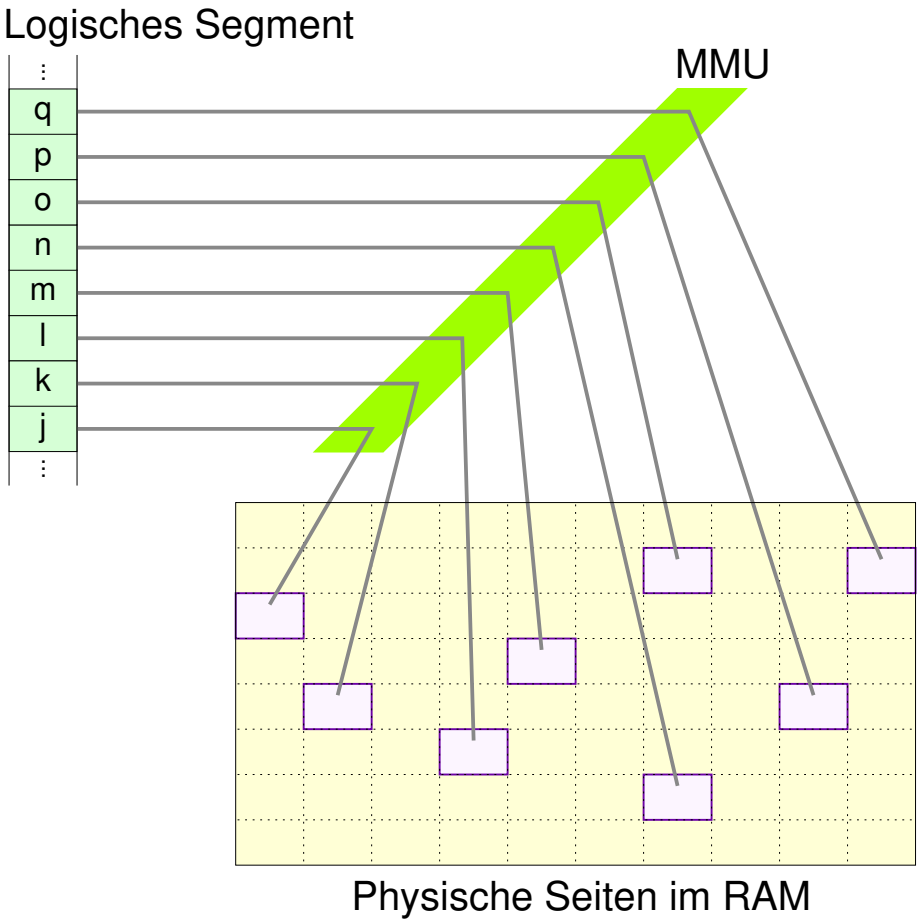


Abbildung 13.6.: Logisches Segment aus physischen Seiten

13.2.2. Kosten

Der Einsatz von MMUs bringt Nachteile beim Speicherverbrauch und bei der Ausführungsgeschwindigkeit mit sich. Zunächst müssen alle Segmente auf vollen Seiten vergrößert werden, obwohl der zusätzliche Platz für Code, Konstanten oder statischen Daten nicht genutzt werden kann. Die Übersetzungstabellen brauchen zusätzlich Speicher, ihre Pflege kostet Rechenzeit. Beides steht für andere Zwecke nicht mehr zur Verfügung.
Besonders kritisch ist das Nachschlagen der Übersetzungen in den Tabellen. Müssten die MMUs für jeden Speicherzugriff des Prozessors eine Tabelle durchsuchen, würden Zugriffszeiten auf ein Vielfaches steigen. Nur durch Caches für Tabelleneinträge bleibt dieser Aufwand im Rahmen. In der Praxis nimmt man die unvermeidlichen Kosten meist in Kauf. Bei allen Geräten, auf denen Anwendungen oder Dienste installiert werden, also ab Smartwatch aufwärts, überwiegen die Vorteile der höheren Systemstabilität.

13.3. Übersetzungslogik

Abschnitt 13.3.1 erläutert kurz den Aufbau von Adressen und führt zwei wichtige Begriffe ein, Seitennummer und Offset. Diese verwende ich in den folgenden Erklärungen. Abschnitt 13.3.2 beschreibt den Aufbau von Übersetzungstabellen bzw. ihre Einträgen. Wo Tabellen liegen und wie MMUs sie finden ist Thema von Abschnitt 13.3.3. Zuletzt räumt Abschnitt 13.3.4 die Vorstellung ab, dass Übersetzungstabellen statische Strukturen sind.

13.3.1. Struktur einer Adresse

Auf dem Systembus oder in einem Prozessor werden Adressen auf einem Bündel von Adressleitungen übertragen. In jeder Leitung fließt Strom in der einen oder anderen Richtung, was einem Bit an Information entspricht. Wir können diese Information als 0 oder 1 interpretieren. Die Adressleitungen sind geordnet von höher- zu niederwertigen Bits. Die 0en und 1en in geordneter Reihenfolge ergeben den Zahlenwert der Adresse im Binärsystem. Weil Menschen mit langen Sequenzen von 0en und 1en schlecht umgehen können, fasst man jeweils vier Bit zu einer Hex–Stelle zusammen. Diesen Weg der Interpretation zeigt Abbildung 13.7 an einem Beispiel mit 32 Adressbits. Eine Umrechnung von Adressen in Dezimalzahlen ist möglich, aber hier nutzlos.

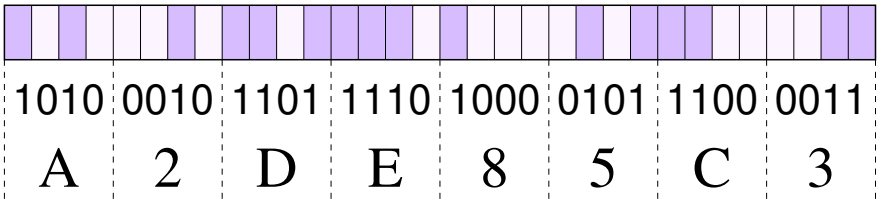


Abbildung 13.7.: Adresse, binär und hexadezimal

Beim Übergang zu Seiten teilt man Adressen an einer festen Position auf, wie in Abbildung 13.8 dargestellt. Drei Hexstellen entsprechen 12 Adressbits. Diese erlauben $2^{12} = 4096$ verschiedene Adressen. An jeder Adresse liegt ein Byte des Speichers, so ergibt sich eine Seitengröße von 4 Kilobyte (KiB). Für eine Seitengröße von 64 Kilobyte würde man 16 Bit, also vier statt drei Hexstellen, abtrennen. Die höherwertigen Bits links bilden die *Seitennummer*, die abgetrennten, niederwertigen Bits rechts den *Offset* (dt.: Versatz) innerhalb der Seite. Die Anfangsadresse jeder Seite liegt an Offset 0.

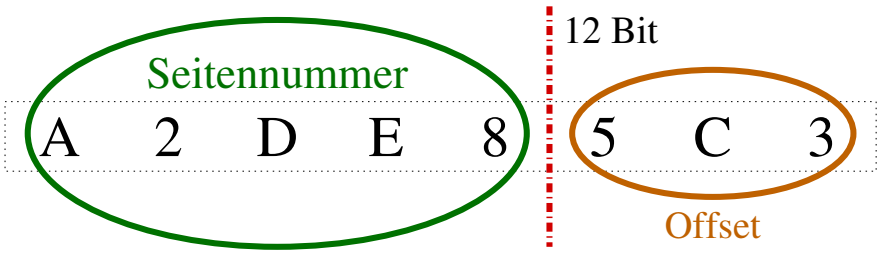


Abbildung 13.8.: Seitennummer und Offset

13.3.2. Struktur einer Tabelle

Eine MMU übersetzt ganze Seiten von logischen in physische Adressen. Anders ausgedrückt übersetzt sie nur die logische Seitennummer in eine physische, wie in Abbildung 13.9 dargestellt. Der Offset einer übersetzten Adresse bleibt gleich, er läuft an der MMU vorbei.

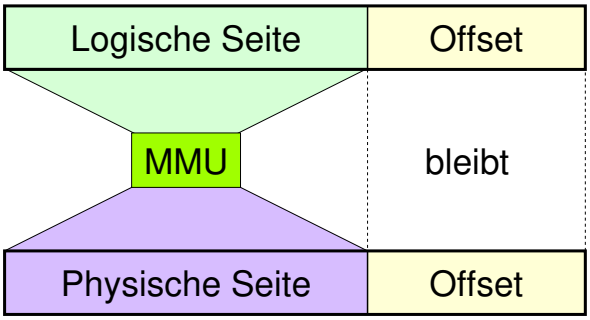


Abbildung 13.9.: Die MMU übersetzt nur Seitennummern

Jeder Eintrag in einer Übersetzungstabelle legt für genau eine logische Seite(nnummer) die entsprechende physische fest. Zudem enthält der Eintrag Flags, welche die erlaubten Zugriffsarten definieren. Typische Zugriffsarten sind:

Code lesen, für Code–Segmente

Daten lesen, für Konstanten–Segmente

Daten schreiben und lesen, für Stapel, Halde und statische Daten

Der Prozessor übermittelt an die MMU nicht nur die Seitennummer auf den Adressleitungen. Zusätzliche Leitungen zeigen an, ob der Prozessor liest oder schreibt, und ob er auf Code oder Daten zugreift. Außerdem kann der Betriebsmodus des Prozessors eine Rolle spielen. Dieser ändert sich, wenn ein Befehlsstrom des Kerns statt einer Instanz läuft. Die

Logische Seite	Physische Seite	Zugriffsart
00100 00101 ⋮	… …	Code lesen
02000 02001 ⋮	… …	Daten nur lesen
30000 30001 ⋮	… …	Daten lesen und auch schreiben

Abbildung 13.10.: Struktur einer Übersetzungstabelle

MMU gleicht alle Informationen mit den Flags im Übersetzungseintrag für die logische Seite ab, um zu entscheiden, ob der Zugriff gültig ist. Falls sie keinen Übersetzungseintrag findet, ist der Zugriff offensichtlich ungültig. Meldet die MMU einen ungültigen Zugriff, so stoppt der Prozessor die Ausführung des laufenden Befehlsstroms durch eine Unterbrechung.
Jeder Übersetzungseintrag gilt nur für eine logische Seite. Um ein Segment aus mehreren Seiten zu übersetzen, braucht man also einen Eintrag pro Seite. Daraus ergibt sich die Struktur einer Übersetzungstabelle in Abbildung 13.10. Im Beispiel sind drei Segmente angedeutet, je eines für die drei typischen Zugriffsarten.

13.3.3. Lage der Tabellen

Übersetzungstabellen sind groß. Ein Segment von einem Megabyte Größe braucht 256 Tabelleneinträge, von einem Gigabyte 262.144. Bei 8 oder 16 Byte pro Eintrag liegen die Tabellen selbst im Bereich vom Megabyte. Daten dieses Umfangs passen nicht in Register, sie müssen im Hauptspeicher liegen. Beim Zugriff auf die Tabellen nutzen die MMUs physische Adressen. Das Betriebssystem muss also ausreichend große, zusammenhängende Bereiche des physischen Speichers für die Übersetzungstabellen reservieren.
Jede MMU hat Konfigurationsregister. Über diese wird eingestellt, ob die MMU überhaupt aktiv ist und falls ja, wo ihre Übersetzungstabelle liegt. Das genaue Format der Übersetzungstabellen und ihrer Einträge hängt von der Prozessorarchitektur und Adressbreite ab. Das ist Bitgefiesel auf einem Detailgrad, der hier ablenken statt zum Verständnis beitragen würde.
Abbildung 13.11 zeigt ein Beispiel mit drei Prozessen von zwei verschiedenen Instanzen. Auf dem ersten Prozessor rechnet ein Prozess der Instanz 1, die MMU dort verwendet die Übersetzungstabelle 1. Auf zwei anderen Prozessoren rechnen gleichzeitig verschiedene Prozesse der Instanz 3. Beide MMUs verwenden die gleiche Übersetzungstabelle der Instanz 3. Im RAM liegen eine weitere Übersetzungstabelle der Instanz 2, von der gerade kein Prozess rechnet.

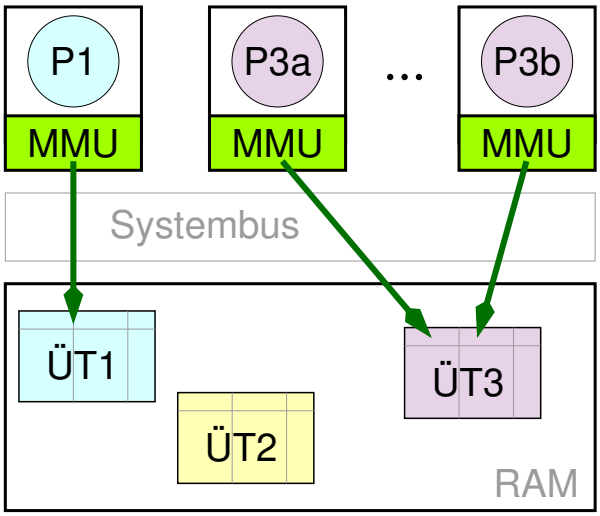


Abbildung 13.11.: Übersetzungstabellen im RAM

Wenn ein Segment von mehreren Instanzen gemeinsam verwendet wird, braucht man Übersetzungseinträge in alle betroffenen Tabellen. Es wäre aber aufwendig und ineffizient, vielfache Übersetzungseinträge parallel zu pflegen. Die MMUs aktueller Architekturen arbeiten mehrstufig, wobei jede Stufe nur einen Teil der Adressbits übersetzt. Das hält unter anderem die Größe der Übersetzungseinträge klein. Frühe Stufen arbeiten mit Seitengrößen von Mega- oder Gigabyte. Erst die letzte Stufe verwendet die oben beschriebene Seitengröße von 4 oer 64 Kilobyte.
Für jede Stufe gibt es eigene Übersetzungstabellen. So ist es möglich, aus verschiedenen Tabellen der frühen Stufen auf eine gemeinsame Tabelle der letzten Stufe zu verweisen. Dort stehen die Übersetzungseinträge für die Seiten des gemeinsam verwendeten Segments. An einer zeichnerischen Darstellung dieses Vorgehens versuche ich mich nicht.

13.3.4. Dynamische Tabellen

Nach der Beschreibung oben könnte man den Eindruck gewinnen, dass Übersetzungstabellen statische Strukturen sind. Der Kern muss sie nur ändern, wenn Segmente angefordert, verändert oder freigegeben werden. Diese Art der Verwendung ist möglich, aber nicht üblich.[2] Normalerweise verwalten Betriebssysteme die Tabellen dynamisch, sie enthalten nur einen Teil der möglichen Übersetzungseinträge. Fehlende Einträge werden bei Bedarf nachgereicht, länger nicht mehr verwendete im Gegenzug entfernt. Die Übersetzungstabelle wird zu einem Cache der zuletzt verwendeten Einträge. Das hält die Größe der Tabellen im Zaum und vermeidet lange Wartezeiten beim Anfordern großer Segmente, in denen noch kein Inhalt steht.
Wenn eine Übersetzungstabelle nicht alle Einträge enthält, dann löst die MMU auch bei einem korrekt ablaufenden Befehlsstrom einen Seitenfehler (engl.: *page fault*) aus, sobald sie eine Adresse nicht übersetzen kann. In der Unterbrechungsbehandlung oder nachgelagerter Logik entscheidet dann der Kern, wie es weitergeht. Drei Fälle kommen in Frage:

Kleiner Seitenfehler: Der Zugriff war gültig, der Speicherinhalt steht in einer physischen Seite bereit, nur der Übersetzungseintrag fehlte. Der Kern aktualisiert schleunigst die Übersetzungstabelle, dann läuft der unterbrochene Befehlsstrom weiter.

Großer Seitenfehler: Der Zugriff war eigentlich gültig, aber der Speicherinhalt steht *nicht* in einer physischen Seite bereit. Der Kern muss erst einmal den Speicherinhalt beschaffen. Bis dahin wird der unterbrochene Prozess blockiert. Kapitel 14 und Kapitel 19 besprechen verschiedene Beispiele für diesen Fall.

Fataler Seitenfehler: Der Zugriff war ungültig, der Befehlsstrom kann nicht weiterlaufen.

Um die Größe der Tabellen zu begrenzen muss der Kern Einträge löschen oder überschreiben. Das sollten möglichst nicht diejenigen sein, die der laufende Befehlsstrom gerade verwendet. Zugriffe auf den Code und den Stapel, aber auch andere Datenzugriffe, erfolgen häufig auf die gleiche oder benachbarte Seiten. Wurde eine Seite dagegen länger nicht mehr verwendet, wird sie wahrscheinlich auch noch ein Weilchen länger nicht mehr verwendet.

[2] Bei COSY für PowerPC [3] bin ich so vorgegangen. Die MMU war nur einer von vielen Aspekten meiner Diplomarbeit. Deshalb musste ich mich auf eine Minimallösung beschränken.

Damit der Kern erkennen kann, welche Einträge überhaupt und auf welche Weise verwendet wurden, setzen MMUs zwei Flags in denjenigen Einträgen, die sie verwenden:

Zugriffsindikator bei jeder Übersetzung

Modifikationsindikator beim Übersetzen eines Schreibzugriffs

Eine Komponente im Kern oder in einem systemnahen Dienst muss regelmäßig die Einträge durchschauen und diese Flags zurücksetzen. Ist ein Flag gesetzt, wurde der Eintrag seit dem letzten Zurücksetzen verwendet und sollte besser in der Tabelle bleiben.

13.4. Knackpunkte

- Der Hauptspeicher erkennt physische Adressen (engl.: *real addresses*) auf dem Systembus. Befehlsströme nutzen logische Adressen (engl.: *virtual addresses*) auf einem Prozessor. Zwischen Prozessor und Systembus übersetzt eine MMU die Adressen.

- Die Übersetzung erfolgt seitenweise. Eine typische Seitengröße ist 4 Kilobyte, also 12 Adressbits. Ein Eintrag in einer Übersetzungstabelle enthält…

  Logische Seitennummer für die der Eintrag gilt
  Physische Seitennummer auf die der Eintrag übersetzt
  Flags für erlaubte Zugriffsarten: Lesen, Schreiben, Code ausführen
  Indikatoren ob eine MMU diesen Eintrag verwendet hat

- Jeder Adressraum, d.h. jede Instanz und auch der Kern, hat eine eigene Übersetzungstabelle im Hauptspeicher. Die MMU eines Prozessors verwendet jeweils die Tabelle des Befehlsstroms, den der Prozessor gerade ausführt. Laufen mehrere Befehlsströme in einem Adressraum gleichzeitig auf verschiedenen Prozessoren, verwenden deren MMUs die gleiche Tabelle.

- Befehlsströme können nur auf Adressen zugreifen, die die MMU übersetzt. Lässt sich eine Adresse nicht übersetzen, löst die MMU eine Unterbrechung aus. Das nennt man Seitenfehler.

- Übersetzungstabellen enthalten nur einen Teil der gültigen Einträge, weil sie sonst zu groß würden. Bei einem *kleinen* Seitenfehler ergänzt der Kern den fehlenden Eintrag, dann läuft der Befehlsstrom weiter. Bei einem *großen* Seitenfehler blockiert der Kern den unterbrochenen Prozess, weil es länger dauert, den Eintrag zu erstellen. Bei einem ungültigen Zugriff stoppt der Kern den Befehlsstrom bzw. Prozess.

- Speicherschutz durch MMUs funktioniert nur auf Ebene der Seiten und Segmente. Ungültige Zugriffe innerhalb eines Segments, zum Beispiel auf einen nicht angeforderten Bereich der Halde, kann eine MMU nicht verhindern.

- Bei gemeinsam verwendeten Segmenten zeigen Übersetzungseinträge in verschiedenen Tabellen auf die gleichen physischen Seiten des Hauptspeichers.

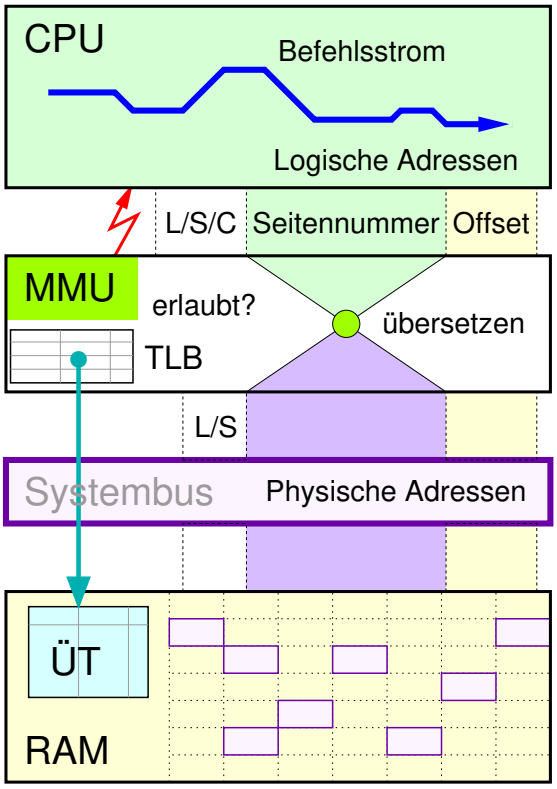


Abbildung 13.12.: Speicherzugriff per MMU mit Details

Abbildung 13.12 kombiniert mehrere, bereits bekannte Abbildungen und ergänzt Details aus dem Text. Auf dem Prozessor oben (CPU) arbeitet ein Befehlsstrom (blauer Pfeil) mit logischen Adressen. Zur MMU darunter gelangen neben den logischen Adressen auch Informationen zur Zugriffsart (L/S/C). Die Konfiguration der MMU zeigt (türkiser Pfeil) auf eine Übersetzungstabelle (ÜT), welche unten im Hauptspeicher (RAM) liegt. Der Cache für Übersetzungseinträge in der MMU heißt *Translation Lookaside Buffer* (TLB).
Die MMU sucht für jede logische Adresse, genauer für deren Seitennummer, den passenden Übersetzungseintrag. Sie kontrolliert auch, dass die Flags im Eintrag die Zugriffsart erlauben. Falls nicht löst sie beim Prozessor eine Unterbrechung aus (roter, gezackter Pfeil). Durch Übersetzen der Seitennummer wird aus der logischen Adresse eine physische, die auf den Systembus gelangt. Als Zugriffsart spielen dort nur noch Lesen oder Schreiben eine Rolle. Der Hauptspeicher ist überwiegend in physische Seiten aufgeteilt, die mittels übersetzter logischer Adressen angesprochen werden. Auf die Übersetzungstabelle greift die MMU direkt mit physischen Adressen zu.

14. Übersetzungstabellen

Der grundlegende Aufbau eines Adressraums, in den eine überschaubare Anzahl zusammenhängender Segmente eingeblendet sind, ist aus Kapitel 3 bekannt. Zur Erinnerung greift Abbildung 14.1 die Darstellung aus Abbildung 3.6 auf Seite 40 wieder auf. Um die MMU zur Speicherverwaltung und Zugriffskontrolle einzusetzen, muss der Kern für jeden Adressraum eine passende Übersetzungstabelle bauen. Grundlage dafür sind die von der Kernschnittstelle unterstützten Kernobjekte zur Speicherverwaltung. In COSY [3] waren das Adressräume, Segmente und Einblendungen als jeweils eigene Kernobjekte. Ich bespreche im Folgenden diese API, weil sie das Zusammenspiel der Informationen besonders deutlich macht. In heute gebräuchlichen Betriebssystemen gibt es andere, aber vergleichbare Wege, um Adressräume zu gestalten.

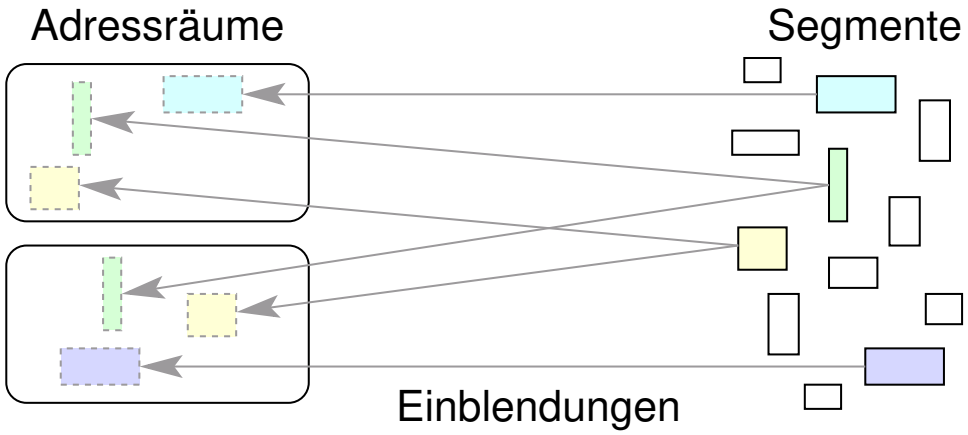


Abbildung 14.1.: Bausteine für Adressräume

In Abschnitt 14.1 erläutert die Informationen, die der Kern verwalten muss, um Übersetzungstabellen zu erstellen. Abschnitt 14.2 beschäftigt sich mit der Größe von Übersetzungstabellen, und wie man diese beschränkt. Abschnitt 14.3 beschreibt den Umgang mit Seitenfehlern durch fehlende Einträge in einer Tabelle. Abschnitt 14.4 fasst die wichtigsten Punkte dieses Kapitels zusammen.

14.1. Verwaltung

Jedes Segment hat eine bestimmte Größe. Da Zugriffe durch die MMU gehen sollen, muss diese Größe ein Vielfaches der Seitengröße sein. Außerdem dient jedes Segment einem bestimmten Zweck. Es enthält zum Beispiel Code, Konstanten, eine Halde oder einen Stapel. Zum Erstellen einer Übersetzungstabelle muss der Kern wissen, ob Code- oder nur Datenzugriffe, ob Schreib- oder nur Lesezugriffe auf das Segment erlaubt sein sollen. In COSY gab es dafür Flags am Kernobjekt, die der Programmlader passend setzen konnte. Da ein Segment Speicherinhalte bereitstellt, muss der Kern auch wissen, wo diese Inhalte liegen. Im einfachsten Fall ist das eine Liste physischer Seitennummern. Diese Information bleibt Instanzen verborgen, man kann sie nicht über die Kernschnittstelle abfragen.
Durch eine Einblendung gelangt ein bestimmtes Segment in einen bestimmten Adressraum. Dabei erhält es eine Basisadresse in diesem Adressraum. Die Basisadresse bestimmt die logische Seitennummer, ab der das Segment im Adressraum zugreifbar ist. All Seiten des Segments erscheinen direkt hintereinander, an aufeinanderfolgenden logischen Seitennummern, wie in Abschnitt 13.2 beschrieben. Die Basisadresse muss so gewählt sein, dass sich die Einblendungen im Adressraum nicht überlappen. Jede logische Seite darf nur einmal in der Übersetzungstabelle stehen, deshalb muss die Einblendung eindeutig sein. In COSY hatte die Einblendung neben der Basisadresse auch noch einmal Flags, wie schon das Segment. So konnte man ein schreibbares Segment in manche Adressräume schreibbar, in andere nur lesbar einblenden.

	Adressraum	**Einblendung**	**Segment**
API	-	Basisadresse, Flags	Größe, Flags
Kern	-	-	Lage des Inhalts
MMU	Übersetzungstabelle	Einträge in der ÜT	

Abbildung 14.2.: Verwaltungsdaten für Adressräume

Die Tabelle in Abbildung 14.2 fasst diese Informationen zusammen. Jede Spalte entspricht einem Kernobjekttyp. An der API des Kerns sind Basisadresse, Größe und Flags sichtbar. Zudem sollte es noch die Möglichkeit geben, alle Einblendungen eines Adressraums bzw. eines Segments abzufragen. Diese Objektbeziehungen sind in der Tabelle nicht dargestellt. In der zweiten Zeile stehen interne Attribute der Kernobjekte, nämlich die Lage des Inhalts beim Segment. Wie oben erwähnt ist das im einfachsten Fall eine Tabelle mit einer physischen Seitennummer für jede Seite des Segments. Kompliziertere Fälle kommen weiter unten zur Sprache.
Die unterste Zeile der Tabelle führt auf, welche Informationen der Kern für die MMU bereitstellen muss. Für jeden Adressraum gibt es eine eigene Übersetzungstabelle. Bei einem Adressraum ohne Einblendungen ist diese leer. Mit jeder Einblendung gelangen Übersetzungseinträge für die Seiten des Segments in die Tabelle. Die verschiedenen Teile eines Übersetzungseintrags ermittelt der Kern aus den Attributen von Einblendung und Segment:

- Logische Seitennummer aus der Basisadresse der Einblendung
- Physische Seitennummer aus der Lage des Inhalts des Segments
- Zugriffsflags aus den Flags von Einblendung und Segment.

Im Rahmen meiner Diplomarbeit [3] hatte ich COSY für PowerPC tatsächlich so implementiert, dass beim Erzeugen einer Einblendung sofort alle Übersetzungseinträge erstellt wurden. Das war möglich, weil der Hauptspeicher mit 8 MB, also 2048 physischen Seiten, relativ klein war und Effizienz bei der Portierung nur eine geringe Rolle spielte. Der folgende Abschnitt 14.2 beschreibt einen anderen, praxistauglichen Umgang mit Übersetzungseinträgen.

14.2. Tabellengröße

Mit Hauptspeicher in GB–Größe ist es nicht praktikabel, vollständige Übersetzungstabellen aufzubauen. Angenommen ein Programm startet mit einer Halde von 1 GB (2^{30}). Mit einer Seitengröße von 4 KB (2^{12}) umfasst dieses Segment 262.144 Seiten (2^{18}). Bei einer Größe von 8 Byte pro Übersetzungseintrag müssten beim Start des Programms 2 MB an Einträgen generiert und in die Übersetzungstabelle geschrieben werden. Und das alleine für die Halde, die zu diesem Zeitpunkt noch leer ist.
Praktisch dienen die Übersetzungstabellen als Cache für diejenigen Einträge, die tatsächlich gebraucht werden. Wenn eine Instanz mit einer leeren Übersetzungstabelle startet, meldet die MMU zunächst in schneller Folge Übersetzungsfehler. Dann trägt der Kern jeweils die fehlende Übersetzung in die Tabelle ein. Nach einiger Zeit sind die häufig benötigten Einträge verfügbar. Um die Größe der Tabelle zu beschränken, wird man über kurz oder lang auch alte Tabelleneinträge löschen bzw. mit neuen überschreiben. Der Zugriffsindikator hilft dabei, länger nicht mehr verwendete Einträge zu finden.
Wenn ein Segment in mehrere Adressräume eingeblendet ist, müssten für dessen physische Seiten eigentlich mehrere Einträge in verschiedenen Übersetzungstabellen erstellt werden. Das führt zu mehr Verwaltungsdaten. Außerdem wird es dadurch aufwendiger, Zugriffs- und Modifikationsindikatoren für eine physische Seite zu bestimmen. Moderne MMUs bieten deshalb die Möglichkeit, in verschiedene Tabellen gemeinsame Untertabellen einzubinden. So kann der Kern für jedes Segment eine eigene Untertabelle anlegen und diese in den Übersetzungstabellen verschiedener Adressräume referenzieren.

14.3. Seitenfehler

Wenn die MMU eine Adresse nicht übersetzen kann, löst sie einen Seitenfehler aus. Aufgrund der unvollständigen Übersetzungstabellen passiert das nicht nur bei Programmierfehlern, sondern auch bei korrekt arbeitenden Instanzen. Der Kern muss jeweils die Ursache bestimmen und passende Maßnahmen einleiten. Dabei kann man grob drei Kategorien von Seitenfehlern unterscheiden:

- Ein ungültiger Zugriff führt zu einem fatalen Seitenfehler. Der Prozess darf nicht weiterlaufen. Falls die Instanz keine Fehlerbehandlung für diesen Fall eingerichtet hat, wird sie abgeschossen.
- Ein *kleiner* Seitenfehler liegt vor, wenn ein Übersetzungseintrag fehlt und schnell nachgereicht werden kann. Der Prozess läuft danach weiter. Idealerweise wird er nicht einmal blockiert.
- Ein *großer* Seitenfehler erfordert aufwendige Maßnahmen zur Behebung. Der Prozess blockiert, bis diese abgeschlossen sind.

Der typische kleine Seitenfehler entsteht durch einen fehlenden Übersetzungseintrag. Die physische Seite ist bekannt und im Segment hinterlegt, der Kern kann den Eintrag umgehend erstellen. Eine andere Art von kleinem Seitenfehler entsteht beim ersten Zugriff auf eine leere Seite. Oben habe ich das Beispiel einer großen, leeren Halde beim Programmstart beschrieben. Den gesamten Speicher der Halde beim Start anzufordern und mit Nullen zu füllen bedeutet erheblichen Aufwand, der die Startzeit verlängert. Statt dessen kann man im Segment hinterlegen, dass alle Seiten leer sind. Die Lage des Inhalts in der Tabelle aus Abbildung 14.2 ist also „nirgendwo“. Gleichzeitig sollte der Bedarf an leeren Seiten vorgemerkt werden, aber ohne konkrete physische Seiten anzufordern. Diese verzögerte Anforderung kann man bei Halde, Stapeln und den nicht oder null–initialisierten statischen Daten einsetzen.
Im Kern muss es einen Pool von leeren, also mit Nullen gefüllten Seiten geben, der regelmäßig aufgefüllt wird. Beim ersten Zugriff auf eine leere Seite nimmt der Kern eine Seite aus diesem Pool, trägt sie im Segment ein und erstellt den fehlenden Übersetzungseintrag. Das dauert kaum länger als wenn die physische Seite schon vorher bekannt ist. Falls der Pool gerade keine leeren Seiten mehr enthält, handelt es sich um einen großen Seitenfehler. Der Prozess blockiert, bis eine leere Seite für ihn verfügbar ist. Weitere große Seitenfehler beschreiben die folgenden Unterabschnitte kurz.

14.3.1. Copy–on–Write

In Linux, Unix und verwandten Betriebssystemen ist es gängige Praxis, eine laufende Instanz durch Aufruf von `fork()` zu duplizieren. Dabei entsteht ein neuer Adressraum, der zunächst den exakt gleichen Inhalt hat wie der ursprünglich. Erst bei Schreibzugriffen laufen die Inhalte auseinander.
Statt alle schreibbaren Segmente bei `fork()` zu kopieren, vermerkt der Kern in den alten und neuen Segmenten, dass die Seiten erst bei einem Schreibzugriff kopiert werden sollen. Alle Übersetzungseinträge stellt er zunächst auf nur lesbar. Somit führen Schreibzugriffe zu einem Seitenfehler. Dann muss der Kern eine leere (aber nicht mit Nullen gefüllte) Seite belegen, den Inhalt der ursprünglichen Seite hineinkopieren und einen neuen, schreibbaren Übersetzungseintrag erzeugen. Alle anderen Adressräume, die sich diese Seite noch teilen, bleiben unverändert.

14.3.2. MMIO

In Linux, Unix und anderen Betriebssystemen ist es möglich, den Inhalt einer Datei als Segment in den Adressraum einzublenden. Dieses Verfahren heißt *memory–mapped IO* (MMIO). Bei einem Zugriff auf das Segment muss der Kern erst den passenden Teil der Datei in eine physische Seite laden. So lange bleibt der Prozess blockiert. Bei Schreibzugriffen sollte der Speicherinhalt auch regelmäßig wieder in die Datei zurückgeschrieben werden. Dabei hilft der Modifikationsindikator in den Übersetzungseinträgen. Ein wenig mehr zu MMIO steht in Abschnitt 19.4 auf Seite 186 unter „Dateizugriff“.

14.3.3. Virtueller Speicher

Beim sogenannten *Paging* werden selten gebrauchte Inhalte des Hauptspeichers auf Festplatte oder SSD ausgelagert. Die neue Lage des Inhalts wird im Segment hinterlegt, Übersetzungseinträge gelöscht und die nun freien physischen Seiten für andere Zwecke verwendet. Bei einem Zugriff müssen Inhalte zunächst wieder in den Speicher geladen werden. Die Mechanismen zum Laden und Auslagern sind prinzipiell die gleichen wie bei MMIO. Mehr zu virtuellem Speicher steht in Kapitel 19.

14.4. Knackpunkte

- Kernobjekte zur Verwaltung des Hauptspeichers sind Adressräume, Segmente und Einblendungen.
- An der Kernschnittstelle sichtbar sind die Größe des Segments und die Basisadresse der Einblendung. Beide sind Vielfache der Seitengröße. Flags zur Zugriffskontrolle können sowohl am Segment als auch an der Einblendung hängen.
- Einblendungen in den gleichen Adressraum dürfen sich nicht überlappen.
- Das gleiche Segment kann in verschiedenen Adressräumen an unterschiedlichen Basisadressen eingeblendet sein.
- Zu einem Segment muss der Kern wissen, wo dessen Speicherinhalte liegen. Diese Information bleibt vor Instanzen verborgen.
- Instanzen greifen ausschließlich über logische Adressen auf ihre Speicherinhalte zu. Eine MMU übersetzt logische Adressen in physische, oder löst Seitenfehler aus.
- Zu jedem Adressraum gehört eine eigene Übersetzungstabelle.
- Einträge in der Übersetzungstabelle kombinieren Informationen von Einblendung und Segment. Die Einblendung liefert logische und das Segment physische Seitennummern. Zugriffsflags können von beiden stammen.
- Vollständige Übersetzungstabellen wären zu groß. Deshalb steht jeweils nur ein Teil der möglichen Übersetzungseinträge in der Tabelle.
- Ein kleiner Seitenfehler liegt vor, wenn ein fehlender Übersetzungseintrag umgehend nachgereicht wird. Der Prozess rechnet weiter.
- Ein großer Seitenfehler liegt vor, wenn der Speicherinhalt gerade nicht in einer physischen Seite zur Verfügung steht. Der Prozess blockiert, bis die Daten vorliegen.
- Initial leere Segmente brauchen noch keine physischen Seiten. Erst beim Zugriff auf eine Seite muss der Kern eine mit Nullen gefüllte, physische Seite bereitstellen.
- MMIO (engl.: *memory–mapped IO*) blendet eine Datei als Segment in einen Adressraum ein. Der erste Zugriff auf jede Seite führt zu einem großen Seitenfehler, der den entsprechenden Teil der Datei einliest.
- *Paging* lagert ungenutzte Speicherinhalte auf einen persistenten Massenspeicher aus. Jeder Zugriff auf ausgelagerte Inhalte führt zu einem großen Seitenfehler, der die entsprechenden Daten wieder einliest.

16. Kernaufrufe

In Kapitel 12 wurde das Umschalten zwischen Prozessen heruntergebrochen auf Kerneinund -austritte. Kurze Befehlsströme von Kernaktionen unterbrechen die langlebigen Befehlsströme der Prozesse. Die Kernaktionen blockieren oder deblockieren Prozesse und geben dem Aufgreifer Gelegenheit, einen neuen Prozess zu wählen, der Rechenzeit bekommt. Im vorliegenden Kapitel geht es nun speziell um die Vorgänge beim Kernein- und -austritt. Außerdem liefere ich ein paar Details zur Arbeitsweise eines Kerns nach, die für Instanzen keine Rolle spielen.

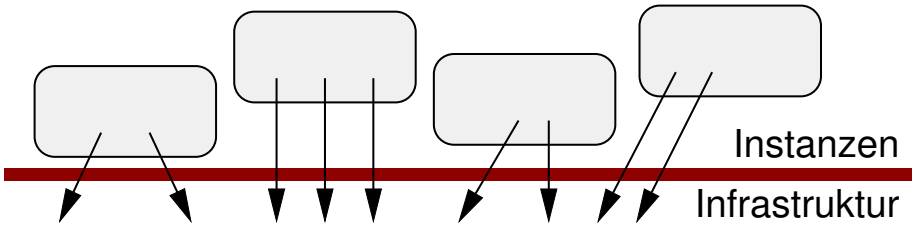


Abbildung 16.1.: Instanzen rufen den Kern

Jeder Kerneintritt ist eine Unterbrechung (engl.: *interrupt*) auf einem der Prozessoren im Rechner. Unterbrechungen treten häufig auf, hunderte pro Sekunde sind üblich. Unterbrochen wird für gewöhnlich ein langlebiger Befehlsstrom, also ein Prozess. Damit einher geht ein Wechsel des Prozessors in den Systemmodus. Entsprechend ist jeder Kernaustritt das Ende einer Unterbrechungsbehandlung, verbunden mit einem Wechsel zurück in den Anwendungsmodus. Unterbrechungen von Unterbrechungsbehandlungen sind ein Sonderfall, den man zu vermeiden sucht.
Abschnitt 16.1 beschreibt Kerneintritte aus der Sicht von Instanzen. Abschnitt 16.2 wechselt zur Perspektive des Kerns. Er erläutert den Umgang mit den Prozessorregistern zu Beginn und Ende jeder Kernaktion. Das sind die entscheidenden Schritte beim Umschalten von Prozessen. Abschnitt 16.3 geht darauf ein, wie der Kern selbst langlebige oder mittellange Befehlsströme nutzen kann. Abschnitt 16.4 fasst die wichtigsten Punkte dieses Kapitels zusammen.

16.1. Kerneintritte

Um den Kern aufzurufen, löst ein Prozess durch einen speziellen Befehl eine Unterbrechung aus, einen Software–Interrupt. Solche Kerneintritte finden also an bestimmten, vorab bekannten Stellen des Befehlsstroms statt. Sie sind *synchron* zum Ablauf des Prozesses. Der Befehlsstrom stellt Parameter bereit, bevor er den Kern aufruft. Sobald er nach der Unterbrechung wieder fortsetzt, kann er mit den Ergebnissen arbeiten, die der Kernaufruf

liefert. Aus Sicht des Prozesses ist die aufgerufene Kernoperation *atomar*. Es gibt einen Zustand vor dem Aufruf und einen nach dem Aufruf. Aber der aufrufende Befehlsstrom kann nicht auf einen Zwischenzustand treffen, in dem die Kernoperation noch läuft und Daten inkonsistent sind.
Die meisten anderen Unterbrechungen erfolgen *asynchron*. Ein Befehlsstrom kann nicht wissen, an welcher Stelle er zum Beispiel durch eine abgelaufene Zeitscheibe oder einen Peripheriechip unterbrochen wird. Es gibt weder Parameter noch Ergebnisse. Deshalb kann die ausgelöste Kernaktion nicht mit dem unterbrochenen Prozesses interagieren. Vielmehr darf der Befehlsstrom nichts von der Unterbrechung bemerken, etwa in den Registern des Prozessors. Die Kernaktion kann nur den Zustand des unterbrochenen Prozesses ändern, ihn zum Beispiel zurückstellen oder blockieren.
Die Erklärungen in Kapitel 12 zu einzelnen Kernaktionen und zusammengesetzten Kernoperationen gelten sowohl für synchrone als auch für asynchrone Kerneintritte. Der Unterschied liegt nur in den fehlenden Parametern und Ergebnissen im asynchronen Fall. Alle Beispiele in Kapitel 10 sind synchrone Kerneintritte, weil die Methoden der Kernobjekte explizit aufgerufen werden müssen.

Anmerkung: *Ein Sonderfall des synchronen Kerneintritts sind Befehle, die der Prozessor nicht kennt. Das kann passieren, wenn ein Programm Befehle einer neueren Prozessorgeneration enthält, aber auf einem älteren Prozessor abläuft. Dann könnte der Kern die neuen Befehle emulieren, statt die Ausführung des Programms abzubrechen.*

Für synchrone Kerneintritte gilt eine Aufrufkonvention, so wie bei Aufrufen innerhalb eines übersetzten Programms. Parameter und Ergebnisse liegen in bestimmten Registern oder auf dem Stapel. Manche Register darf der Kern verändern. Allerdings erzeugen Compiler von sich aus keinen Code, der Unterbrechungen auslöst, die den Kern aufrufen. Dazu braucht es Handarbeit in Form von Assemblercode.
Um Kernoperationen bequem aufrufen zu können, steckt man diesen Assemblercode in eine Bibliothek, deren Schnittstelle ganz normale Funktionen oder Methoden anbietet. Diese laden übergebene Parameter in die richtigen Register, erstellen die vom Kern erwarteten Datenstrukturen und lösen dann den Kerneintritt aus. Anschließend geben sie die Ergebnisse des Kernaufrufs in einem aufruferfreundlichen Format zurück. Eine solche Bibliothek adaptiert die Aufrufschnittstelle des Kerns, sein ABI (engl.: *Application Binary Interface*), in eine Aufrufschnittstelle der jeweiligen Programmiersprache, eine API (engl.: *Application Programming Interface*). Das ist ausführlich in Abschnitt 6.2 beschrieben.

16.2. Umgebungswechsel

Alle Befehlsströme, sowohl die von Prozessen als auch die des Kerns, arbeiten mit den Registern der Anwendungssicht aus Abbildung 15.1 auf Seite 147. In Kernaktionen sind insbesondere die Ganzzahl- und Sprungregister im Einsatz, während mathematische Berechnungen mit Gleitkommaregistern oder Befehlssatzerweiterungen eher selten sind. Speicherzugriffe sowohl auf Code als auch auf Daten hängen zusätzlich von der Konfiguration der MMU des Prozessors ab.

Befehlsströme von Instanzen laufen im Anwendungsmodus des Prozessors. Sie können nur die Register der Anwendungssicht verändern. Die Konfiguration der MMU ist ihnen vom Kern vorgegeben. Befehlsströme des Kerns dagegen laufen im Systemmodus des Prozessors. Der Kern hat also Zugriff auf die Register der Systemsicht aus Abbildung 15.2 auf Seite 149. Damit kann er unter anderem die Konfiguration der MMU ändern, also die Übersetzungstabelle manipulieren. Das sollte allerdings nur auf dem jeweiligen Prozessor passieren und nicht für MMUs auf anderen Prozessoren sichtbar sein, die eventuell gerade die gleiche Übersetzungstabelle verwenden.
Für die folgenden Betrachtungen teile ich den Registersatz eines Prozessors in drei Gruppen auf:

Rechenumgebung: Alle schreibbaren Register der Anwendungssicht, mit Befehls- und Stapelzeiger. Alle Befehlsströme nutzen diese Register für Berechnungen. Parameter und Ergebnisse von Aufrufen liegen in der Rechenumgebung oder auf dem Stapel.

Ablaufumgebung: Register der Systemsicht, die den Ablauf des aktuellen Befehlsstroms beeinflussen. Das ist insbesondere die Konfiguration der MMU. Der Kern kann diese Register ändern, Instanzen nicht.

Systemumgebung: Die restlichen Register der Systemsicht. Damit sie auch einen Namen haben, selbst wenn der nie wieder auftaucht.

Damit ein Befehlsstrom korrekt abläuft, müssen auf einem Prozessor sowohl die Rechen- als auch die Ablaufumgebung für diesen Befehlsstrom geladen sein. Beim Wechsel zwischen Befehlsströmen sind die Umgebungen kurzzeitig inkonsistent. Eben dies passiert beim Kernein- und -austritt, weshalb hier besondere Vorsicht geboten ist.
Abbildung 16.2 zeigt den Ablauf einer Kernaktion ohne Umschalten, also ein Detail aus Abbildung 12.2 auf Seite 124. Der gelbe Mittelteil steht für die eigentliche Kernaktion, die meist in einer übersetzten Programmiersprache wie C geschrieben ist. Davor und danach sind zusätzlich die notwendigen Umgebungswechsel markiert. In diesen Bereichen kommt Assembler zum Einsatz.

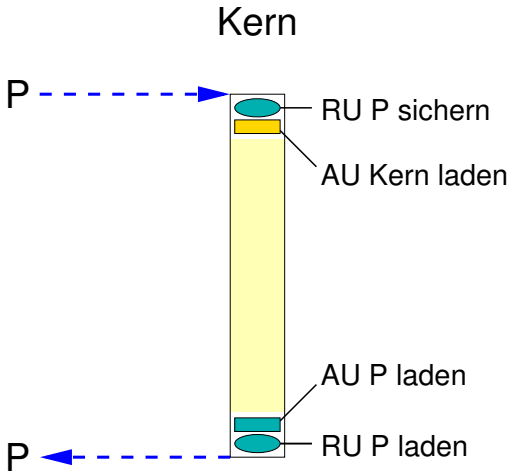


Abbildung 16.2.: Umgebungswechsel ohne Umschalten

Der Befehlsstrom beginnt durch eine Unterbrechung des laufenden Prozesses. In der Rechenumgebung stehen noch die Werte dieses Prozesses. Nur den Befehlszeiger hat der Prozessor an anderer Stelle gesichert, um nun die Unterbrechungsbehandlungsroutine auszuführen. Die Ablaufumgebung ist ebenfalls für den unterbrochenen Prozess initialisiert. Allerdings schalten Unterbrechungen die MMU ab. Mit der Unterbrechung erfolgte außerdem der Wechsel vom Anwendungsmodus des Prozessors in den Systemmodus.
Da der unterbrochene Prozess (P) später fortgesetzt werden soll, sichert der Befehlsstrom zuerst dessen Rechenumgebung (RU). Damit sind diese Register zur weiteren Verwendung frei. Anschließend lädt er die Ablaufumgebung (AU) des Kerns. Das bedeutet typischerweise, dass er die MMU wieder einschaltet. Entweder nur mit der Übersetzungstabelle des Kerns, oder mit einer Kombination der Tabellen des unterbrochenen Prozesses und des Kerns. Jedenfalls kann die Kernaktion auf den Speicher des Kerns zugreifen. Die Ablaufumgebung des Prozesses zu sichern wäre überflüssig, denn der Prozess konnte sie nicht ändern.
Der Hauptteil der Kernaktion beginnt nach einem asynchronen Kerneintritt mit einer „leeren“ Rechenumgebung, in der nur Befehls- und Stapelzeiger initialisiert sind. Nach einem synchronen Kerneintritt, stehen in der Rechenumgebung Aufrufparameter, mit denen die Kernaktion arbeitet. In diesem Fall gilt eine Aufrufkonvention zwischen Prozess und Kern. Das bedeutet, dass der Prozess Änderungen an seiner Rechenumgebung erwartet. Dann muss beim Kerneintritt auch nur ein Teil der Register gesichert werden, was Zeit spart.
Zum Ende hin, nach Ausführung der eigentlichen Kernaktion, stellt der kurze Befehlsstrom die Ablaufumgebung des Prozesses wieder her bzw. bereitet das vor. Das kann bedeuten, dass die MMU wie beim Kerneintritt abgeschaltet wird, damit Zugriffe auf die gesicherte Rechenumgebung mit physischen Adressen arbeiten. Die Übersetzungstabelle des Prozesses wird in der MMU konfiguriert. Anschließend lädt der Befehlsstrom die Rechenumgebung des Prozesses, mit Ausnahme des Befehlszeigers. Bei einem synchronen Kernaufruf wurden zuvor Rückgabeparameter in die gespeicherte Rechenumgebung eingetragen. Der allerletzte Befehl, welcher die Unterbrechungsbehandlung beendet, lädt dann den Befehlszeiger des Prozesses und schaltet die MMU mit der vorbereiteten Übersetzungstabelle ein. Damit erfolgt auch der Wechsel zurück vom System- in den Anwendungsmodus des Prozessors.

Abbildung 16.3 zeigt den Ablauf einer Kernaktion mit Umschalten, also ein Detail aus Abbildung 12.3 auf Seite 125. Zu Beginn verläuft der Befehlsstrom ebenso wie zuvor beschrieben. Das kann auch nicht anders sein, denn dass die Kernaktion zu einem anderen Prozess umschaltet, stellt sich erst im Verlauf heraus. Auf den Mittelteil hat das ebenfalls noch keinen Einfluss. Die von einem Compiler übersetzte Kernaktion läuft normal zu Ende. Erst bei den Vorbereitungen zum Rücksprung aus der Unterbrechungsbehandlung wirkt sich aus, dass nun ein anderer Prozess zum Zuge kommt, der vielleicht in einem anderen Adressraum läuft. Statt nur Änderungen rückgängig zu machen, muss dann eine komplett andere Übersetzungstabelle in die Ablaufumgebung geladen werden. Das Laden der Rechenumgebung des neuen Prozesses wird auch aufwendiger. Dabei sind nun Register zu berücksichtigen, die der Kern selbst nicht verwendet und die deshalb bei einer Kernaktion ohne Umschalten weder gesichert noch neu geladen werden.

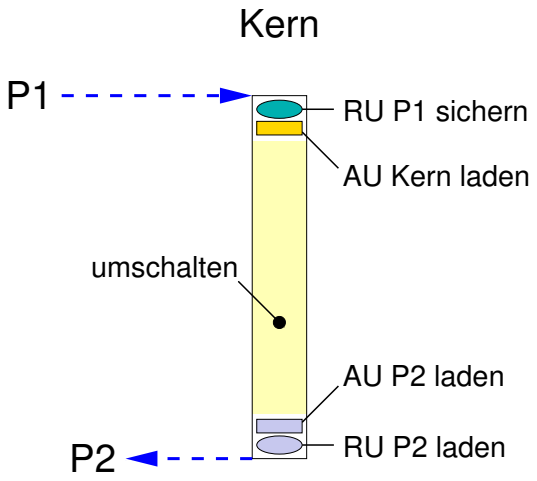


Abbildung 16.3.: Umgebungswechsel mit Umschalten

In Anlehnung an Elefantenwitze gibt Abbildung 16.4 eine Merkhilfe für den Umgang mit den Registerumgebungen. Die Antworten auf „Wie bekommt man einen Elefanten in den Kühlschrank?“ und „Wie bekommt man eine Giraffe in den Kühlschrank?“ setze ich als bekannt voraus.

Frage: Wie bekommt man einen Eisbären in den Kühlschrank?

Antwort:

1. Tür auf
2. Giraffe raus
3. Temperatur runterdrehen
4. Eisbär rein
5. Tür zu

Abbildung 16.4.: Eselsbrücke für die Prozessumschaltung

16.3. Prozesse im Kern

Bisher habe ich eine zweistufige Arbeitsweise des Betriebssystems vorgestellt. Im Instanzenbereich kommen langlaufende Prozesse zum Einsatz, die Rechenzeit vom Kern erhalten und bei Bedarf blockieren. Der Kern selbst erledigt seine Aufgaben in Unterbrechungsbehandlungen, die schnell ablaufen und nicht blockieren können. Das war der Ansatz von COSY für PowerPC.[3] Er ist aber nur bedingt praxistauglich. Zwei miteinander zusammenhängende Probleme sprechen gegen diese Struktur.
Zum einen müssen Unterbrechungsbehandlungen sehr schnell ablaufen, damit der Prozessor unverzüglich auf Ereignisse im System reagieren kann. Schon das Umkopieren weniger Kilobyte im Rahmen einer Sendeoperation dauert länger, als es für eine Unterbrechungsbehandlung angemessen wäre. Kernoperationen mit den Speicherobjekten aus Kapitel 14 können dazu führen, dass größere Mengen Speicher gelöscht und viele Einträge in Übersetzungstabellen erstellt oder entfernt werden müssen. Es ist nicht realistisch, solche Operationen in Unterbrechungsbehandlungen zu erledigen.
Zum anderen verfügen moderne Rechnern über mehrere Prozessoren, die auch gleichzeitig im Kern arbeiten können sollen. Da Unterbrechungsbehandlungen nicht blockieren dürfen ist es schwierig, sie zu koordinieren, wenn sie auf die gleichen Datenstrukturen des Kerns zugreifen müssen. Je länger Unterbrechungsbehandlungen dauern, desto mehr Rechenleistung geht verloren, weil Prozessoren auf andere warten müssen.

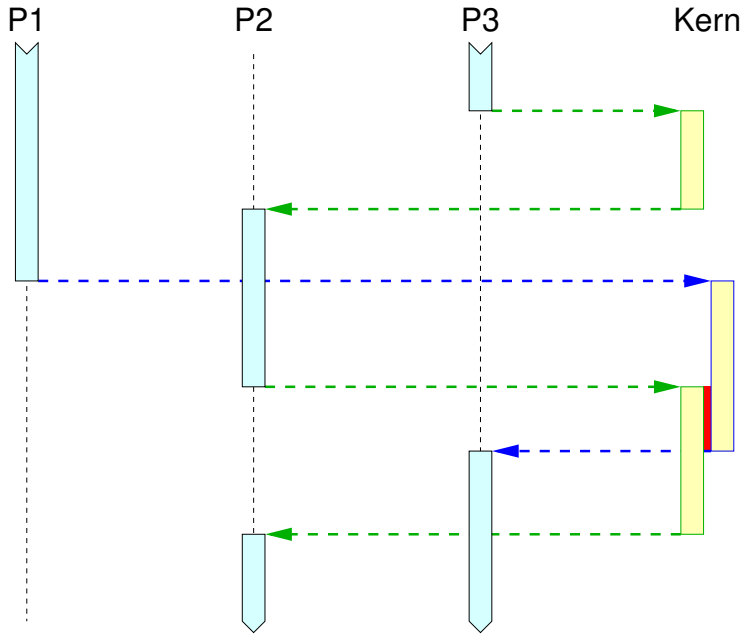


Abbildung 16.5.: Kernaktionen bei zwei Prozessoren

Abbildung 16.5 zeigt einen möglichen Verlauf von Kernaufrufen mit zwei Prozessoren. Die Kernaktionen eines Prozessors sind mit blauen Pfeilen eingezeichnet, die des anderen mit grünen. Zu Beginn arbeiten die Prozessoren an P1 und P3. Wenn die Prozesse in verschiedenen Adressräumen ablaufen, sind sie voneinander unabhängig. Laufen sie im gleichen Adressraum, muss sich der Anwendungsentwickler um die Konsistenz seiner Daten kümmern.

Dann wird P3 auf dem grünen Prozessor durch einen Kernaufruf unterbrochen, der zu P2 umschaltet. Während der Kernaktion arbeiten P1 und der Kern parallel. Der Kern greift nur auf den Adressraum von P1 zu, falls eine zuvor aufgerufene Kernoperation das verlangt. Auch in diesem Fall muss sich der Anwendungsentwickler darum kümmern, dass P1 keinen Kernoperationen aus seinem Adressraum in die Quere kommt.
Nach dem Umschalten des grünen Prozessors arbeiten P1 und P2 parallel. Das ist wieder die Ausgangssituation, nur mit anderen Prozessen. Auch der folgende Kerneintritt des blauen Prozessors führt zu einer bekannten Lage, der Kern arbeitet parallel zu einem Prozess. Kritisch wird es mit dem zweiten Kerneintritt des grünen Prozessors, während die Kernaktion des blauen Prozessors noch läuft. Jetzt arbeiten zwei Prozessoren im Kernadressraum und können auf die gleichen Datenstrukturen zugreifen, zum Beispiel die Bereitmenge. In diesem Fall muss die Systementwicklerin dafür Sorge tragen, dass die Daten des Kerns konsistent bleiben und alle Kernaktionen korrekt ablaufen. Ein Weg dazu sind Sperren. Allerdings kann eine Unterbrechungsbehandlung nicht blockieren und den Prozessor für andere Aufgaben freigeben, während sie auf eine Sperre wartet. Statt dessen verwendet man *Spinlocks* (dt.: Drehsperren), bei denen der Prozessor in einer Dauerschleife ständig die Sperre prüft, bis er sie setzen kann.
Um Unterbrechungsbehandlungen im Kern kurz zu halten, gibt es zwei naheliegende Möglichkeiten. Einerseits kann man zeitaufwendige Operationen an Instanzen delegieren. Dazu richtet man spezielle Kernschnittstellen ein, damit Dienstprozesse unter anderem die Verwaltung von Speicherseiten, die Pflege von Übersetzungstabellen oder das Umkopieren von Daten im Speicher übernehmen. Das bedeutet allerdings zusätzlichen Verwaltungsaufwand, mehr Schnittstellen und ein komplexeres Gesamtsystem. Das Abschießen einer Instanz etwa erfordert vielleicht die Mitwirkung mehrerer Dienstprozesse.
Die zweite Möglichkeit ist es, lang laufende Operationen mit Prozessen innerhalb des Kerns zu bearbeiten. Diese Prozesse verwenden dann zur Sicherung von Datenkonsistenz ganz normale Sperren. Im Gegensatz zu Unterbrechungsbehandlungen können Prozesse blockieren, auch im Kern. So entsteht eine dreistufige Arbeitsweise des Betriebssystems:

1. Prozesse in Instanzen
2. Prozesse im Kern
3. Unterbrechungsbehandlung im Kern

Linux verwendet sowohl Kernprozesse (engl.: *kernel threads*), die nur im Kernadressraum arbeiten, als auch Instanzenprozesse, die für einen Kernaufruf vorübergehend in einen Kernmodus (engl.: *kernel mode*) umschalten. Abbildung 16.6 zeigt den Ablauf eines Kernaufrufs in diesem Fall. Sie ist das Gegenstück zu Abbildung 12.2 auf Seite 124, wo eine Unterbrechungsbehandlung die gesamte Kernaktion erledigt. Auch in Linux erfolgt der Kernaufruf durch Unterbrechungen.[1] Der blaue Befehlsstrom des Prozesses bleibt ausgesetzt, bis die Kernoperation beendet ist. Die Unterbrechungsbehandlung führt hier aber keine Kernoperation aus, weder ganz noch teilweise. Statt dessen schaltet sie den aufrufenden Prozess in den Kernmodus. In diesem Modus führt der Prozess einen Befehlsstrom des

[1] Linux x86 Kernel Entries, Version 5.11.0-rc5, abgerufen 2021-01-28:
`https://www.kernel.org/doc/html/latest/x86/entry_64.html`

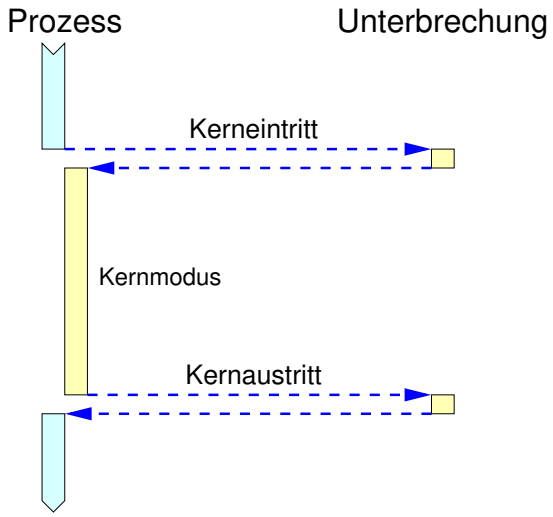


Abbildung 16.6.: Prozess im Kernmodus

Kerns aus, der die übergebenen Argumente prüft, die gerufene Operation ausführt und die Ergebnisse bereitstellt.
Der Befehlsstrom des Kerns beginnt mit dem Wechsel in den Kernmodus und endet mit dem Wechsel zurück in den Instanzmodus (engl.: *user mode*). Im Gegensatz zu Kernaktionen in Unterbrechungsbehandlungen kann dieser Befehlsstrom des Kerns jedoch länger laufen und auch blockieren, ohne dass dies zu Problemen führt. Den Kernaustritt habe ich in Abbildung 16.6 aus Gründen der Symmetrie als Unterbrechungsbehandlung dargestellt. Wenn der Prozess während der Kernoperation im Systemmodus des Prozessors arbeitet, ist auch ein direkter Wechsel zurück aus dem Kernmodus denkbar. Allerdings müsste das sehr sorgfältig programmiert werden, damit die Verwaltungsinformation des Prozesses konsistent mit dem gerade aktiven Befehlsstrom bleibt.

16.4. Knackpunkte

- Kerneintritte sind Unterbrechungen.
- Kernaufrufe sind synchron ausgelöste Unterbrechungen, mit Parametern und Ergebnissen in Registern.
- Jeder Befehlsstrom arbeitet mit den Registern seiner Rechenumgebung (RU). Das gilt für langlebige ebenso wie für kurze Befehlsströme.
- Jeder Befehlsstrom braucht seine Ablaufumgebung (AU), seine Übersetzungstabelle in der MMU. Ein Prozess kann seine AU nicht selbst ändern.
- Zu Beginn sichert eine Kernaktion die RU des unterbrochenen Befehlsstroms. Seine AU liegt unverändert in den Daten des Kerns.
- Am Ende lädt eine Kernaktion AU und RU des folgenden Befehlsstroms.
- Eine Kernaktion kann den folgenden Befehlsstrom ändern, also zu einem anderen Prozess umschalten.

- Umschalten geht nur zwischen Prozessen, also langlebigen Befehlsströmen. Unterbrechungsbehandlungen, also kurze Befehlsströme, müssen zu Ende laufen.
- Zeitaufwendige Aktionen gehören nicht in Unterbrechungsbehandlungen.
- Verbreitete Betriebssysteme nutzen Prozesse im Kern für zeitaufwendige Aktionen.
- Für Prozesse in Instanzen spielt es keine Rolle, ob Kernaufrufe von Unterbrechungsbehandlungen oder von Prozessen im Kern erledigt werden.

17. Dateisysteme

Nachdem Kapitel 5 die Navigation durch Verzeichnisstrukturen zum Thema hatte, wenden wir uns nun Dateisystemen auf Datenträgern zu. Dabei bleibe ich allerdings auf einer allgemeinen Ebene, welche die meisten gängigen Dateisysteme abdeckt. Bitgenaue Datenstrukturen konkreter Implementierungen spielen weiterhin keine Rolle. Abschnitt 17.1 beschreibt die grundlegenden Objekte, die den Inhalt eines Dateisystems modellieren. Abschnitt 17.2 ergründet die Beziehungen zwischen Dateisystemen und Datenträgern. Abschnitt 17.3 erläutert einige spezielle Dateisysteme, welche nicht in das zuvor beschriebene Schema passen. Dazu gehören Pseudo– und Netzwerk–Dateisysteme. Abschnitt 17.4 fasst die wichtigsten Punkte dieses Kapitels zusammen.

17.1. Organisation

Die meisten Datenträger präsentieren sich als eine Folge von Blöcken gleicher Größe, zum Beispiel 4 KB. Der Zugriff erfolgt über Blocknummern. Ein Dateisystem für solche Datenträger muss Nutz- und Metadaten sowie interne Verwaltungsdaten in Blöcken unterbringen. Die nach außen sichtbaren Objekte zur Verwaltung von Nutzdaten sind aus Kapitel 5 bekannt:

Verzeichnisse mit potentiell vielen Einträgen

Dateien mit potentiell umfangreichen Nutzdaten

Symlinks oder *Reparse Points* mit überschaubarer Größe

17.1.1. Inodes

Der Begriff Inode (engl.: *index node*) stammt aus dem Unix–Umfeld und bezeichnet eine interne Datenstruktur eines Dateisystems. Jedes Verzeichnis, jede Datei, jeder Symlink belegt einen Inode. Jeder Verzeichniseintrag verweist auf den Inode des verzeichneten Objekts. Die Inodes eines Dateisystems auf einem Datenträger tragen eine Nummer, die innerhalb des konkreten Dateisystems eindeutig ist. Die Nummer kann man abfragen, um zu prüfen, ob Objekte identisch sind. Ansonsten referenziert man Objekte ausschließlich über Pfade, die beim Zugriff zu einem Inode aufgelöst werden. Die Inode–Nummern bilden eine interne Kennung für Objekte.
Im Inode steht, wo auf dem Datenträger die Nutzdaten des Objekts liegen. Bei einer kleinen Datei oder einem leeren Verzeichnis können die Nutzdaten direkt mit im Inode untergebracht sein. Bei großen Dateien oder Verzeichnissen mit vielen Einträgen führt der Inode zu umfangreichen Datenstrukturen, die bei Bedarf Millionen von Blöcken des Datenträgers referenzieren.

Anmerkung: *Dateisysteme ohne Unix–Hintergrund müssen die gleichen Probleme lösen und verwenden ähnliche Datenstrukturen. Ausnahmen bilden veraltete Dateisysteme für Disketten mit geringer Speicherkapazität und nur lesbare Dateisysteme. Mit solcherlei Einschränkungen geht es auch einfacher.*

17.1.2. FAT

Aus dem Windows–Umfeld stammt eine Familie von Dateisystemen, welche eine *File Allocation Table* (FAT) statt Inodes verwenden. Der Datenträger wird in Cluster unterteilt, die aus einer festen Anzahl von Blocks bestehen. Vereinfacht enthält die FAT für jeden Cluster einen Eintrag, in dem die Nummer des folgenden Clusters steht.[1] So lassen sich beliebig viele Cluster miteinander verketten. Jeder Verzeichniseintrag muss auf den ersten Cluster des Objekts verweisen. Für Dateien steht auch die genaue Länge im Verzeichnis. Symbolische Links und harte Links sind nicht vorgesehen. Für große Datenträger verschwendet dieses Verfahren viel Platz für kleine Dateien. Die Variante exFAT wird weiterhin für SD–Karten verwendet, auf denen eher große Dateien mit Videos und Fotos landen. [2]

17.1.3. Metadaten

Die meisten aktuellen Dateisysteme pflegen Metadaten für Dateien und andere Objekten. Dazu gehören die Zeitpunkte, an denen das Objekt erzeugt und zuletzt verändert wurde. Auch der letzte Zugriff kann mitgeführt werden, was allerdings zu Schreibzugriffen auf das Dateisystem führt, selbst wenn man Nutzdaten nur liest. Zugriffsrechte sind Teil der Metadaten in POSIX–Dateisystemen. Das schließt die Kennungen des Benutzers und der Gruppe ein, denen ein Objekt gehört.
In Dateisystemen ohne harte Links kann man Metadaten in den Verzeichniseinträgen unterbringen. Bei FAT stehen dort die Zeitstempel, Zugriffskontrolle gibt es nicht. Mit harten Links können aber mehrere Verzeichniseinträge für die gleiche Datei existieren. Deshalb stehen dort nur Name und Inode im Verzeichniseintrag, andere Metadaten im Inode. Ebenfalls im Inode steht, wieviele Verzeichniseinträge auf ihn verweisen. So erkennt das System, ob eine Datei endgültig gelöscht wurde.
Die genannten Beispiele sind traditionelle, feste Bestandteile aller POSIX–Dateisysteme. Moderne Dateisysteme, auch in POSIX, erlauben frei definierbare, erweiterte Attribute (engl.: *Extended Attributes*).[3] So kann man zum Beispiel einen MIME–Typ für den Dateiinhalt hinterlegen, oder feingranulare Zugriffsrechte (ACL[4], engl.: *Access Control List*). Allerdings besteht die Gefahr, dass erweitere Attribute beim Kopieren oder Überschreiben verloren gehen.

[1] `https://www.cs.virginia.edu/~cr4bd/4414/S2021/slides/20210413-slides.pdf` abgerufen 2023-02-19
[2] `https://www.youtube.com/watch?v=bikbJPI-7Kg` 12 Minuten abgerufen 2023-02-19
[3] `https://man7.org/linux/man-pages/man7/xattr.7.html` abgerufen 2023-02-19
[4] `https://linux.die.net/man/5/acl` abgerufen 2023-02-19

17.1.4. Verwaltung

Für Lesezugriffe genügen die oben beschriebenen Informationen. Der in Abschnitt 5.1 erwähnte Wurzelblock (engl.: *root block*) liegt an einer festen Position des Datenträgers. Dort steht der Inode des obersten Verzeichnisses, über das die Inodes aller weiteren Verzeichnisse, Dateien und symbolischen Links erreichbar sind. In den Inodes stehen Metadaten und die Blocknummern für Nutzdaten. Im Wurzelblock finden auch Metadaten über das Dateisystem Platz, zum Beispiel ein Name für den Datenträger. Aber meist will man auch noch Daten schreiben können.
Um Platz für Nutzdaten zu finden, müssen die freien Blöcke auf dem Datenträger bekannt oder zumindest auffindbar sein. Defekte Blöcke sollten aus dem Verkehr gezogen werden. Zum Anlegen neuer Dateien, Verzeichnisse oder symbolischer Links braucht es frei Inodes. Inodes können in festen Tabellen verwaltet oder dynamisch erzeugt werden. Das sind Aufgaben, für die verschiedene Dateisysteme unterschiedliche Lösungsansätze wählen, auf die ich hier nicht näher eingehen möchte.
Insbesondere bei Festplatten, mit ihren mechanischen Schreib–Lese–Köpfen, hängen hohe Schreib- und Lesegeschwindigkeiten davon ab, dass Daten möglichst in aufeinanderfolgenden Blöcken liegen. Das sollte die Verwaltung berücksichtigen, wenn freie Blöcke belegt werden. Auf die zur Ablage verwendeten Datenstrukturen kann es sich aber auch auswirken. Die Information "80 Blöcke ab Nummer X"braucht weniger Platz als 80 einzelne Blocknummern.
Ist ein konkretes Dateisystem längere Zeit im Einsatz, entstehen durch gelöschte kleine Dateien oder Verzeichnisse kleine Bereiche freier Blöcke. Das kann dazu führen, dass später große Dateien zerstückelt in diese kleinen Bereich abgelegt werden. Diese Fragmentierung kann wiederum dazu führen, dass Zugriffe spürbar langsamer werden, als sie es aufgrund der Hardware sein müssen. Das *Defragmentieren* ist eine Reorganisation der gespeicherten Daten, um jedes Objekt in aufeinanderfolgende Blöcke zu legen und die freien Blöcke wieder in wenige, große Bereiche zusammenzufassen.
Schreibzugriffe erfordern oft Änderungen an mehreren Stellen im Dateisystem, also auch in verschiedenen Blöcken auf dem Datenträger. Zum Beispiel das Erzeugen einer Datei, die einen Block mit Nutzdaten enthält:

- Einen freien Block finden und als belegt markieren.
- Einen freien Inode finden und als belegt markieren.
- Einen neuen Verzeichniseintrag für die Datei und den Inode anlegen.
- Den Inhalt der Datei in den dafür belegten Block schreiben.

Fällt zwischendrin der Strom aus, geraten die auf dem Datenträger gespeicherten Informationen in einen inkonsistenten Zustand. *Journaling* ist ein Weg, die Auswirkungen abzumildern. Zuerst wird im Journal eingetragen, welche Veränderungen an den verschiedenen Stellen geplant sind. Dann werden die Änderungen durchgeführt, anschließend der Erfolg im Journal vermerkt. Beim Einhängen des Dateisystems erkennt das System, ob dort noch nicht abgeschlossene Änderungen stehen. Dann kann es diese prüfen und entweder zu Ende bringen oder rückgängig machen. Moderne Dateisysteme setzen Journaling von vornherein für die Verwaltungsdaten ein. Ein Journaling der Nutzdaten würde bedeuten, dass diese immer zweimal geschrieben werden müssen, was die effektive Schreibrate halbiert.

17.1.5. Initialisierung

Ein konkrektes Dateisystem ist eine komplizierte Datenstruktur, welche zunächst initialisiert werden muss. Das *Formatieren* eines Datenträgers erfüllt diese Aufgabe. Dabei werden der Wurzelblock, interne Verwaltungsstrukturen und ein leeres Verzeichnis angelegt, auf das der Wurzelblock zeigt. Anschließend sind Schreibzugriffe möglich, um Nutzdaten im neuen Dateisystem abzulegen. Zum Formatieren muss bekannt sein, wieviel Platz der Datenträger bietet. Davon hängen die Verwaltungsstrukturen ab. Bei modernen Dateisystemen kann man die initial gesetzte Größe auch nachträglich erhöhen. Wozu das gut ist, wird in Abschnitt 17.2 klar. Ein Verringern der Größe ist schwieriger, weil dabei Blöcke verschwinden, in denen bereits Daten liegen können.
Eine zweite Möglichkeit zum Einrichten eines Dateisystem wäre das Einspielen eines zuvor erstellten Backups auf Blockebene. Dabei landen auch gleich Nutzdaten auf dem Datenträger. Allerdings arbeiten Backup–Lösungen selten auf der Blockebene, sondern speichern nur die Nutz- und Metadaten aus einem Dateisystem. Ein Backup interner Verwaltungsstrukturen und ungenutzter Blöcke würde unnötigen Aufwand verursachen. Außerdem wäre es umständlicher, nur Teile des Backups wieder herzustellen.
Speichermedien wie CD–ROM, die keine Schreibzugriffe auf das Dateisystem erlauben, erfordern ein anderes Vorgehen. Das Dateisystem wird nicht direkt auf dem Speichermedium erzeugt, sondern erst in eine große Datei geschrieben. Im Fall der CD–ROM spricht man von einem ISO–Image, da es auf dem Standard ISO 9660 beruht. Beim *Mastering* des Image müssen sämtliche Nutzdaten und die gewünschte Verzeichnisstruktur bekannt sein. Das vereinfacht die Verwaltungsstrukturen gegenüber einem schreibbaren Dateisystem. Jede Datei liegt am Stück in aufeinanderfolgenden Blöcken, die gesamte Verzeichnisstruktur steht am Anfang des Dateisystems, freie Blöcke gibt es nicht. Nach dem Mastering kann man das Image aus der Datei auf einen oder mehrere Datenträger brennen.

17.2. Blockgeräte

Ein Blockgerät (engl.: *block device*) ist die im Unix–Umfeld gängige Abstraktion für Datenträger oder Massenspeicher. Die in Abschnitt 17.1 vorausgesetzten Zugriffe über lineare Blocknummern erfolgen über diese Abstraktion. So kann die Implementierung eines Dateisystems sowohl mit Festplatten als auch mit USB–Sticks genutzt werden. Die Implementierung dieser Blockgeräte ist unterschiedlich und führt die Zugriffe passend zur Hardware aus.
Die traditionelle Blockgröße war 512 Byte. Mit steigender Kapazität der Messenspeicher ist sie gewachsen und liegt heute typischerweise bei 4 KB. Das ist auch die typische Größe einer Seite im Hauptspeicher, siehe Kapitel 13. Dies ist kein Zufall. Die Übertragung von Daten zwischen Hauptspeicher und Massenspeicher ist einfacher, wenn Blöcke vollständig in eine Seite passen.
Bei den Betrachtungen oben mag der Eindruck entstanden sein, dass ein Dateisystem genau einen Datenträger komplett verwaltet. Das ist keineswegs der Fall, wie ich in den folgenden Abschnitten ausführe. Datenträger können sowohl aufgeteilt als auch zusammengefasst werden. Ein Blockgerät muss auch nicht für einen Datenträger stehen. Das *Loopback device* (dt.: Rückkopplungsgerät) stellt den Inhalt einer Datei als Blockgerät dar. So kann man

zum Beispiel ein ISO–Image als Dateisystem einhängen, ohne es auf eine CD zu brennen. Es vereinfacht auch das Testen oder Ausprobieren von Dateisystemen.
Bei Blockgeräten kann man Blöcke in beliebiger Reihenfolge lesen und ggfs. auch mehrfach überschreiben. Das entspricht dem Verhalten älterer Festplatten und Disketten. Bei direkt angeschlossenem Flash–Speicher ist nur einmaliges Schreiben möglich, aber kein Überschreiben ohne vorheriges Löschen. Dafür gibt es eine andere Abstraktion, und spezielle Dateisysteme. Allerdings wird Flash–Speicher selten direkt angeschlossen. SSDs und USB–Sticks enthalten selbst eine Abstraktionsschicht, die den Inhalt als ein traditionelles Blockgerät erscheinen lässt. Auch bei magnetischen Festplatten mit überlappenden Spuren (engl.: *shingled magnetic recording, SMR*) gibt es ähnliche Probleme und Lösungen. Im Folgenden geht es aber nur um Blockgeräte mit dem traditionellen Verhalten.

17.2.1. Partitionen

Festplatten und andere Massenspeicher lassen sich in *Partitionen* aufteilen. Jede Partition wird selbst zu einem Blockgerät, auf dem man unabhängig von den anderen ein Dateisystem anlegen kann. Auf dem Blockgerät für den gesamten Massenspeicher darf man das dann nicht mehr. Dort wird nur die Partitionstabelle hinterlegt. Zwischen den Partitionen und am Ende kann man Platz lassen, um später noch weitere Partitionen anzulegen oder die davor liegenden Partitionen zu vergrößern. Letzteres ergibt nur Sinn, wenn man danach auch das Dateisystem auf der Partition vergrößert, wie in Abschnitt 17.1.5 angesprochen. Einige Gründe, warum man einen Datenträger partitionieren würde:

Daten mit unterschiedlicher Lebensdauer: Auf meinen selbst eingerichteten Rechnern lege ich die Benutzerverzeichnisse (`/home`) in eine eigene Partition. So kann ich bei Bedarf ein neues Betriebssystem installieren, ohne meine Benutzerdaten zu verlieren. Auch den Inhalt von Datenbanken, eine Mediathek oder Backups würde ich vom Betriebssystem getrennt ablegen.

Notwendigkeit für ein bestimmtes Dateisystem: Die UEFI–Spezifikation (*Unified Extensible Firmware Interface*) verlangt für das Booten von einem Massenspeicher, dass wichtige Dateien in einer Variante des FAT32–Dateisystems abgelegt sind.[5] Dafür genügt eine kleine Partition, während auf dem Rest des Datenträgers besser geeignete Dateisysteme zum Einsatz kommen.

Verwendung ohne Dateisystem: Eine virtuelle Speicherverwaltung braucht Platz auf einem Massenspeicher, um Inhalte des Hauptspeichers dorthin zu verschieben, siehe Abschnitt 19.2. Dafür kann man eine oder mehrere *swap*–Partitionen bereitstellen, statt Dateien in Dateisystemen.

[5] `https://uefi.org/specs/UEFI/2.10/13_Protocols_Media_Access.html#file-system-format`

17.2.2. RAID

RAID steht für *Redundant Array of Inexpensive/Independent Disks*. Mit diesem Verfahren kombiniert man mehrere baugleiche Festplatten zu einer größeren oder schnelleren Gruppe.[6] Dabei kommen verschiedene Modi zum Einsatz:

Striping: Die Blocknummern des Arrays werden stückweise über die verschiedenen Platten verteilt. Die Speicherkapazität ist größer. Schreib- und Lesezugriffe auf große Dateien sprechen verschiedene Platten parallel an, die Übertragungsrate wird größer. Es gibt keine Redundanz.

Mirroring: Der gleiche Inhalt wird parallel auf mehrere Platten geschrieben. Die Speicherkapazität entstpricht der einer einzelnen Platte. Das Schreiben erfolgt parallel und bleibt etwa gleich schnell. Beim Lesen können verschiedene Platten parallel verschiedene Blöcke liefern, die Übertragungsrate wird größer. Es gibt Redundanz. Solange noch eine Platte läuft, sind die Daten verfügbar.

Parität: Ein Teil der Platten wird wie beim Striping zusammengeschaltet, um Kapazität und Übertragungsrate zu erhöhen. Mindestens eine Platte ist für Paritätsbits reserviert, welche über die gleichen Blöcke aller Datenplatten gebildet werden. Das gibt Redundanz, um den Ausfall mindestens einer Platte zu kompensieren. Das Schreiben wird etwas langsamer als beim reinen Striping, weil jeder Schreibzugriff auf eine der Datenplatten auch Schreibzugriffe auf die Paritätsplatten erfordert. Die Paritätsplatten werden zum Flaschenhals.

Verschiedene RAID–Anbieter haben im Lauf der Zeit an Sammelsurium von kombinierten und erweiterten Modi entwickelt. Die Notwendigkeit für baugleiche Festplatten dürfte auch entfallen sein. Allerdings bestimmt weiterhin die kleinste Festplatte in einem Array, wieviel man von allen Platten nutzen kann.

17.2.3. Logical Volume Management

LVM ist eine Abstraktionsschicht, mit der man mehrere Datenträger zu einer *Volume Group* zusammenfassen und innerhalb dieser Gruppe mehrere *Logical Volumes* definieren kann. Jedes Logical Volume wird zu einem Blockgerät, ähnlich wie eine Partition. Im Gegensatz zu Partitionen müssen die Logical Volumes aber nicht kompakt auf einem der Datenträger liegen. Solange in der Volume Group noch irgendwo Platz ist, kann man jedes Logical Volume vergrößeren, und auch weitere anlegen. Wenn kein Platz mehr frei ist, kann man weitere Datenträger zur Volume Group hinzufügen.
Vergrößert man Logical Volumes immer wieder in kleinen Schritten, kann das dazu führen, dass sich alle Logical Volumes über alle Datenträger in der Volume Group erstrecken. Dann beeinträchtigt der Ausfall eines Datenträgers auch alle Logical Volumes. Man sollte deshalb etwas Sorgfalt walten lassen und die Logical Volumes gegebenenfalls umorganisieren, so dass sie auf einem oder wenigen Datenträgern liegen. LVM lässt sich mit RAID kombinieren, um Redundanz zu schaffen. Aber selbst dann sind Backups noch wichtig!

[6] `https://www.ontrack.com/en-gb/blog/30-years-and-counting-will-raid-systems-ever-get-old`

17.3. Spezielle Dateisysteme

17.3.1. Integrierte Dateisysteme

Die Dateisysteme btrfs von Linux und ZFS von Solaris arbeiten nicht nur auf einem einzelnen Blockgerät, sondern integrieren die Funktionen von LVM und RAID, um mehrere Datenträger zu überspannen. Das eröffnet neue Möglichkeiten, um zum Beispiel Metadaten zu spiegeln und Nutzdaten zu stripen. Oder Metadaten auf einem anderen Datenträger zu speichern als die Nutzdaten, damit sich Zugriffe seltener ausbremsen. Außerdem kann ein solches integriertes Dateisystem die gespeicherten Daten umziehen, um zum Beispiel einen Datenträger wieder zu entfernen.

17.3.2. Netzwerk–Dateisysteme

Ein Netzwerk–Dateisystem organisiert nicht die Ablage von Daten auf einem Blockgerät, sondern implementiert ein Protokoll, um auf Dateien und Verzeichnisse eines Servers zuzugreifen. Welche Dateisysteme der Server einsetzt, spielt dabei keine Rolle. Die verbeitetsten Protokolle für diesen Zweck dürften das Network Filesystem (NFS) aus dem Unix–Umfeld sowie Server Message Block (SMB) von Windows sein.

17.3.3. Pseudo–Dateisysteme

Ein gerne zitiertes Prinzip für Unix–Systeme lautet „*everything is a file*“, also alles ist eine Datei. Damit ist gemeint, dass die Mechanismen für Dateizugriffe auch für vieles zum Einsatz kommen, was tatsächlich keine Datei ist. Das funktioniert über Pseudo–Dateisysteme. Sie stellen Systemobjekte in Verzeichnisstrukturen dar, liefern Daten der Objekte bei Lesezugriffen und verändern die Objekte bei Schreibzugriffen. Beispiele aus einem aktuellen Linux–System, ermittelt mit dem Kommando `mount`:

udev ist unter `/dev` eingehängt, wo alle Geräte zu finden sind. Dazu gehören auch die Blockgeräte. Die udev–Komponente erkennt neu angeschlossene oder wieder entfernte Geräte und löst Ereignisse auf, damit Instanzen darauf reagieren können.[7]

proc ist unter `/proc` eingehängt. Es stellt für jede laufende Instanz Informationen bereit, zum Beispiel eine Übersicht der Einblendungen im jeweiligen Adressraum.[8]

sysfs ist unter `/sys` eingehängt und ermöglicht den Zugriff auf interne Objekte des Kerns.[9] In Unterverzeichnissen sind noch weitere Pseudo–Dateisysteme eingehängt.

[7] `https://man7.org/linux/man-pages/man7/udev.7.html`
[8] `https://man7.org/linux/man-pages/man5/proc.5.html`
[9] `https://man7.org/linux/man-pages/man5/sysfs.5.html`

17.4. Knackpunkte

- Dateisysteme bieten nach außen Dateien, Verzeichnisse, Symlinks an. Zu diesen Objekten gehören Metadaten wie Zeitstempel und Zugriffsrechte.
- Intern verwenden Dateisysteme Inode–Nummern, Einträge in einer File Allocation Table oder andere Strategien.
- Datenträger bzw. Massenspeicher sind meist Blockgeräte. Eine typische Blockgröße ist 4 Kilobyte, passend zur Seitengröße des Hauptspeichers.
- Dateisysteme für Blockgeräte organisieren die Ablage der Nutz- und Metadaten in den Blöcken. Für Schreibzugriffe verwalten sie auch freie Blöcke.
- Auf schreibbaren Datenträgern entsteht durch Formatieren ein leeres Dateisystem. Dateisysteme ohne Schreibfunktion erstellt man mit ihrem vollständigen Inhalt.
- Partitionen teilen einen Datenträger in wenige, kompakte Bereiche, die jeweils ein eigenes Dateisystem aufnehmen können.
- RAID fasst mehrere Datenträger zusammen und lässt sie wie einen größeren und/oder schnelleren erscheinen.
- LVM fasst mehrere Datenträger flexibel zusammen und erlaubt die Definition von separaten Bereichen, die nicht kompakt liegen müssen. So kann man Bereiche und die darin liegenden Dateisysteme bei Bedarf vergrößern.
- Netzwerk–Dateisysteme reichen Zugriffe auf Dateien, Verzeichnisse und Symlinks an einen Server weiter.
- Pseudo–Dateisysteme ermöglichen dateiähnliche Zugriffe auf Objekte, die tatsächlich keine Dateien sind. Zum Beispiel auf Blockgeräte.