Zusammenfassung Betriebssysteme

Michael Gregorius

15. November 2002

http://www.michaelgregorius.de/

Vorwort

Diese Zusammenfassung wurde im Rahmen der Vorbereitung auf die zweite Hauptdiplomsprüfung (Inf2) im Studienfach Kerninformatik an der Universität Dortmund erstellt. Die Prüfung erstreckte sich über die Vorlesungen *„Graphische Systeme“* und *„Betriebssysteme“* und wurde am 21. 10. 2002 erfolgreich bei Prof. Dr. Heinrich Müller abgelegt.

Für die Erstellung einer eigenen Zusammenfassung in LaTeX sprachen mehrere Gründe. Zum einen hatte ich schon für einige Prüfungen des Grundstudiums Zusammenfassungen geschrieben und damit sehr gute Erfahrungen gemacht. Denn schon beim Schreiben der Texte lernt man einen großen Teil des Stoffes und da alle Zusammenfassungen, so wie auch diese, in Frage-Antwort-Form geschrieben wurden, wusste man in der Prüfung auch direkt, wie man ein Thema bzw. Themengebiet gut erklären kann. Zum anderen ist das Buch, anhand dessen gelernt wurde, in Englisch geschrieben, so dass es angenehmer war, erst eine deutsche Zusammenfassung davon zu schreiben, anhand derer dann im Anschluss gelernt werden konnte.

Prof. Müller prüft in *„Betriebssysteme“* nach der fünften Auflage des Buchs „Operating System Concepts“ [Sil98] von Abraham Silberschatz. Diese Zusammenfassung wurde größtenteils anhand des inhaltlich sehr ähnlichen Buches „Applied Operating System Concepts“ [Sil00] erstellt, was jedoch kaum ins Gewicht fallen dürfte, da sich die Bücher, wie schon erwähnt, inhaltlich kaum unterscheiden. Dadurch ist es lediglich zu einer unterschiedlichen Kapitelnummerierung gekommen. Die ersten Kapitel wurde teilweise garnicht bzw. recht oberflächlich zusammengefasst, da es in Bezug auf die Prüfung erst ab Kapitel 3 dieser Zusammenfassung interessant wird. Davor werden in den beiden Büchern nur Grundlagen erläutert, die zwar auch nicht unwichtig sind, aber im Verlauf der Bücher meistens nochmal aufgegriffen werden. Die Kapitel 11 und 12 waren bei Prof. Müller nicht prüfungsrelevant. Da ich sie jedoch schon zusammengefasst hatte, bevor ich dies wusste, lasse ich sie in der Zusammenfassung.

Zur Vorlesung *„Graphische Systeme“* wurde ebenfalls eine Zusammenfassung geschrieben, welche, wie die etwas weiter oben schon erwähnten anderen Zusammenfassungen sowie dem Prüfungsprotokoll zu dieser Prüfung, unter `http://www.michaelgregorius.de/` heruntergeladen werden kann. Auf dieser Seite findet sich auch eine Email-Adresse, unter der man mich erreichen kann. Über Kommentare, Meinungen und sonstige Rückmeldungen würde ich mich freuen.

Dortmund, den 11. 11. 2002 *Michael Gregorius*

Inhaltsverzeichnis

1 Einleitung **4**

2 Computersystem-Komponenten **9**

3 Prozesse **13**
3.1 Konzept eines Prozesses 13
3.2 Scheduling von Prozessen 15
3.3 Operationen auf Prozessen 17
3.4 Kooperierende Prozesse 17
3.5 Interprozess-Kommunikation 18

4 Threads **22**

5 CPU Scheduling **25**
5.1 Einleitung 25
5.2 Scheduling-Kriterien 27
5.3 Scheduling Algorithmen 27
5.4 Scheduling mit mehreren Prozessoren 34
5.5 Evaluation von Scheduling Algorithmen 34

6 Prozess-Synchronisation **36**
6.1 Einleitung 36
6.2 Das Critical-Section Problem 37
6.3 Synchronisationshardware 40
6.4 Semaphoren 41
6.5 Klassische Sychronisationsprobleme 44
6.6 Monitore 47

7 Deadlocks **49**
7.1 Einleitung 49
7.2 Characterisierung von Deadlocks 49
7.3 Methoden zum Umgang mit Deadlocks 51
7.4 Deadlock Prevention (Deadlock-Verhinderung) 52
7.5 Deadlock Avoidance (Deadlock-Vermeidung) 53
7.6 Deadlock-Detection 57
7.7 Auflösung von Deadlocks 58

8 Memory Management **60**
8.1 Einleitung 60
8.2 Logischer und physikalischer Adressraum 62
8.3 Swapping 63
8.4 Zusammenhängende Allocation 63
8.5 Paging 65
8.6 Segmentierung 72

9 Virtual Memory **74**
9.1 Einleitung 74
9.2 Demand Paging 74
9.3 Performanz von Demand Paging 76
9.4 Seitenersetzung (Page Replacement) 76
9.5 Algorithmen zur Seitenersetzung 77
9.6 Allokation von Frames 81
9.7 Trashing 82
9.8 Weitere Faktoren 85

10 File Systems **87**
10.1 Dateikonzept 87
10.2 Zugriffsarten 90
10.3 Verzeichnisstrukturen 90
10.4 Schutz des Dateisystems 93
10.5 Struktur eines Dateisystems 94
10.6 Allokationsmethoden 95
10.7 Verwaltung freier Blöcke 99
10.8 Implementierung von Verzeichnissen 101
10.9 Sonstiges 102

11 I/O-Systeme **103**

12 Massenspeicher **117**

13 Verteilte Systeme **123**
13.1 Struktur verteilter Systeme 123
13.2 Verteilte Dateisysteme 125

Abbildungsverzeichnis **129**

Literatur **130**

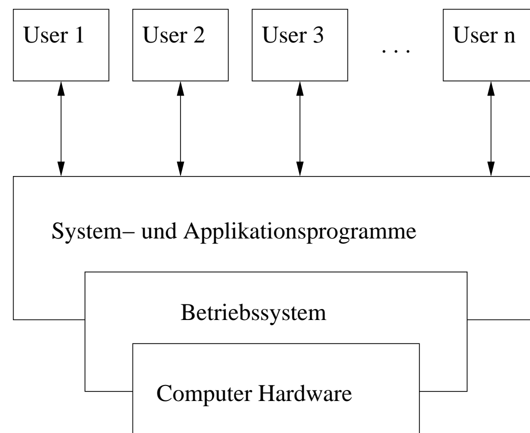


Abbildung 1: Die einzelnen Komponenten eines Computersystems

1 Einleitung

Frage 1.1: Was ist ein Betriebssystem?
Ein Betriebssysteme ist ein Programm welches als Vermittler zwischen dem Benutzer eines Computers und dessen Hardware fungiert. Es verfolgt dabei zwei Ziele:

- Das System soll **in angenehmer Art und Weise** genutzt werden können.
- Die Hardware soll **effizient** genutzt werden.

Diese beiden Ziele können sich teilweise widersprechen, d.h. es gibt oft einen Trade-Off zwischen den beiden. Effizienz ist besonders bei großen Multiuser-Systemen wichtig, da diese oft sehr kostspielig sind.

Es gibt verschiedene Sichtweisen auf ein Betriebssystem. Zum einen ist es ein Resourcen-Zuteiler (**Resource Allocator**). Programme brauchen verschiedene Resourcen (z.B. CPU-Zeit, Hauptspeicher, Ein-/Ausgabe-Geräte) um ihren Zweck zu erfüllen. Eine Aufgabe eines Betriebssystems ist es, dieses Resourcen gerecht zu verteilen.

Eine weitere Sichtweise auf ein Betriebssystem ist, dass es ein Kontroll-Programm (**Control Program**) ist, welches dafür sorgt, dass Programme den Computer in angemessener Weise nutzen und welches Fehler vermeidet.

Frage 1.2: In wieviele Komponenten kann ein Computersystem grob unterteilt werden? Welche sind dies?
Ein Computersystem kann, wie in Abbildung 1 verdeutlicht, grob in vier Komponenten unterteilt werden. Diese sind:

Hardware: Die Hardware stellt die grundlegenden Resourcen zur Lösung von Problemen bzw. Aufgaben zur Verfügung. Dazu zählt z.B. die CPU, der Hauptspeicher, sowie die Ein- und Ausgabegeräte.

Betriebssystem: Das Betriebssytem fungiert als Vermittler zwischen der Hardware und dem Benutzer, sowie den von ihm verwendeten Applikationen.

Man kann sich ein Betriebssystem, wie eine Regierung vorstellen, die für sich selbst genommen nicht nützlich ist, jedoch festlegt, wie die vom Computer zur Verfügung gestellten Resourcen sinnvoll genutzt werden.

Applikationen: Applikationen sind z.B. Textverarbeitungsprogramme, Compiler, Webbrowser oder auch Spiele. Sie legen fest, wie die verschiedenen Resourcen genutzt werden, um die Probleme des Benutzers zu lösen.

Benutzer: In einem System kann es mehrere Benutzer geben, die ihre Aufgaben/Probleme mit Hilfe verschiedener Applikationen lösen möchten.

Frage 1.3: Was ist der Kernel?

Der Kernel ist das Programm, welches zu jeder Zeit auf dem Rechner läuft. Sehr häufig wird der Kernel daher als das Betriebssystem eines Rechners definiert, während alle anderen Programme Anwendungen sind.

Frage 1.4: Welche Arten von (Betriebs-)Systemen gibt es?

- Batch-Systeme
- Time-Sharing-Systeme
- Personal-Computer-Systeme
- Parallel-Systeme
- Real-Time-Systeme (Echtzeit-Systeme)
- Verteilte Systeme

Frage 1.5: Was ist ein Batch-System?

Frühere Computer waren große Maschinen, welche über eine Konsole bedient wurden. Übliche Eingabegeräte waren Kartenleser und Bandlaufwerke. Die Ausgabegeräte bestanden aus Druckern, Bandlaufwerken oder Lochkartenstanzern. Ein Programm wurde üblicherweise in Form von Lochkarten bei dem Operator abgegeben. Auf diesen befand sich das Programm, die Daten, sowie einige Kontrollinformationen. Nach einiger Zeit konnte man das Ergebnis eines Programms beim Operator abholen. Passierte ein Fehler während des Programms bekam man einen **Dump** des Speichers und der Register zurück.

Die einzige Aufgabe des Betriebssystems eines Batch-Systems bestand darin, einen Job nach dem anderen abzuarbeiten. Um den Vorgang zu beschleunigen wurden Jobs mit ähnlichen Anforderungen zusammengefasst und als ein **Batch** in den Computer gegeben. Der Speicheraufbau eines solchen Batch-Systems ist in Abbildung 2 dargestellt.

Frage 1.6: Was versteht man unter Multiprogramming und wie kam es dazu?

Mit der Einführung von Festplatten kam es dazu, dass mehrere Jobs auf der Festplatte in einem **Job Pool** gehalten werden konnten und die Aufgabe des effizienten Ausführens der Jobs auch dem Betriebssystem zufiel (und nicht mehr

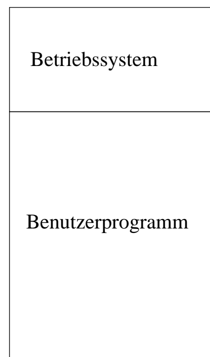


Abbildung 2: Speicherlayout eines einfachen Batch-Systems.

beim Operator lag). Das Betriebssystem konnte also **Job Scheduling** betreiben.

Eine der wichtigsten Eigenschaften des Job Scheduling besteht darin, dass es Multiprogramming ermöglicht. Was bedeutet das? Eine Untermenge der Jobs aus den Job Pool befinden sich im Hauptspeicher. Aus dieser wählt das Betriebssystem jeweils einen aus, der ausgeführt wird. Wartet nun einer dieser Jobs z.B. auf das Ergebnis einer Ein-/Ausgabeoperation, so ist die CPU unbeschäftigt, was nicht erwünscht ist. Beim Multiprogramming wird nun während dieser Zeit einfach der nächste Job ausgeführt, wobei probiert wird, die CPU nie unbeschäftigt zu lassen. Wenn also auch dieser Job wartet, wird der nächste genommen, etc. (Vergleich mit Rechtsanwalt)

Mit dem Aufkommen von Multiprogramming passierte es das erste Mal, dass das Betriebssystem Entscheidungen treffen musste. Da meist nicht alle Jobs auf einmal in den Hauptspeicher passen, muss **Job Scheduling** betrieben werden. Da mehrere Jobs voneinander getrennt im Hauptspeicher sind, muss auch **Memory Management** betrieben werden. Sind mehrere Jobs zur Ausführung bereit, muss weiterhin **CPU Scheduling** betrieben werden.

Frage 1.7: Was ist ein Time-Sharing-System?

Time-Sharing-Systeme (oder auch **Multitasking-Systeme**) sind die logische Erweiterung des Multiprogramming-Konzeptes. In einem interaktiven System kommuniziert der Benutzer in direkter Weise mit dem System, z.B. per Tastatur oder mit der Maus und wartet auf direkte Ergebnisse dieser Kommunikation. Die Antwortzeit (**Response Time**) in einem solchen System sollte dementsprechend gering sein (innerhalb einer Sekunde). In einem Time-Sharing-System wird nun so häufig zwischen den einzelnen Jobs (Prozessen) hin- und hergeschaltet, dass jeder Benutzer direkt mit jedem Programm interagieren kann. Jeder Benutzer bekommt dabei den Eindruck vermittelt, das System stehe nur ihm zur Verfügung.

Frage 1.8: Was ist ein Parallel-System?

Heutige Systeme sind meistens Einprozessor-Systeme, d.h. sie besitzen nur eine CPU. Jedoch gibt es mehr und mehr einen Trend zu Multiprozessor-Systemen. Diese besitzen mehrere Prozessoren, die miteinander kommunizieren und sich

den Bus, den Takt und machmal auch den Hauptspeicher und die Peripherie teilen. Diese Systeme werden auch **Tightly Coupled Systems** genannt.

Frage 1.9: Welche Vorteile haben Parallel-Systeme?

Parallel-Systeme bieten die folgenden Vorteile:

Erhöhter Durchsatz: Durch die Verwendung mehrerer Prozessoren wird erhofft, dass sich diese verschiedene Aufgaben teilen, welche daher schneller bearbeitet werden können. Jedoch arbeiten n Prozessoren nicht n-mal schneller, da durch die Verwaltung der Prozessoren ein gewisser Overhead dazukommt. Desweiteren kann es zu Staus (**Congestion**) kommen, da z.B. der Bus geteilt wird.

Kostenersparnis: Parallel-Systeme können gegenüber mehreren Einprozessor-Systemen günstiger sein, da sie sich Peripherie teilen können.

Erhöhte Verlässlichkeit: Dadurch dass Aufgaben an mehrere Prozessoren verteilt werden, fällt ein System nicht komplett aus, wenn einer der Prozessoren ausfällt. Es wird in diesem Fall einfach nur langsamer. Diese Eigenschaft wird **Graceful Degradation** genannt.

Frage 1.10: Was ist der Unterschied zwischen symmetrischem und asymmetrischen Multiprocessing?

Beim symmetrischen Multiprocessing läuft auf jedem Prozessor eine Kopie des Betriebssystems und diese kommunizieren miteinander. In einem asymmetrischen System wird jedem Prozessor eine bestimmte Aufgabe zugeteilt. Dabei kontrolliert ein Master-Prozessor das System. Die anderen Prozessoren warten entweder auf Anweisungen des Master-Prozessors oder erfüllen eine vorherbestimmte Aufgabe. Auf diese Art und Weise entsteht eine Master-Slave-Beziehung.

Beim symmetrischen Multiprocessing (**SMP**) sind alle Prozessoren gleichberechtigt (**peers**). Da jeder Prozessor eine Kopie des Betriebssystems laufen lässt, könnnen verschiedene Prozesse gleichzeitig laufen. Dabei muss jedoch bei der Ein-/Ausgabe aufgepasst werden, damit die Daten den richtigen Prozessor erreichen. Außerdem kann es passieren, dass ein Prozessor völlig ausgelastet ist, während ein anderer nichts zu tun hat. Durch die Benutzung gemeinsamer Datenstrukturen kann dies jedoch in einem gewissen Rahmen verhindert werden.

Der Unterschied zwischen AMP und SMP kann sowohl durch Hardware als auch durch Software bedingt sein.

Frage 1.11: Was ist ein Real-Time-System (Echtzeit-System)?

Echtzeit-Systeme sind eine Art von speziellen Betriebssystemen. Sie werden überall dort eingesetzt wo strenge Zeitbeschränkungen eingehalten werden müssen. Dazu zählen z.B. industrielle Fertigungsanlagen, wissenschaftliche Experimente, medizinischen Bildgebungsverfahren, Automotoren und Waffen, um nur einige zu nennen. In Echtzeit-Systemen muss eine Berechnung während eines gewissen Zeitrahmens ausgeführt werden oder das System versagt (Beispiel:

Ein Roboterarm, der erst in das Auto, welches er baut, reinfährt und dann den Befehl zum Anhalten bekommt).

Frage 1.12: Welche zwei Arten von Echtzeit-Systemen gibt es?
Es gibt **Hard Real-Time Systems** und **Soft Real-Time Systems**.

Hard Real-Time Systems: Diese Systeme garantieren, dass kritische Aufgaben innerhalb eines festgelegten Zeitrahmens beendet werden. Dazu müssen alle eventuelle Verzögerungen in dem System nach oben beschränkt sein. Dazu gehören alle Anfragen an das Betriebssystem selbst oder auch wie lange es dauert Daten zu holen. Aufgrund dieser Tatsachen fehlen fortgeschrittenen Betriebssystem-Konzepte oft völlig in diesen Systemen.

Soft Real-Time Systems: Diese Systeme sind nicht ganz so streng wie Hard Real-Time Systems. In diesen Systemen bekommen kritische Echtzeit-Aufgaben eine höhere Priorität gegenüber anderen Tasks.

Frage 1.13: Was ist ein verteiltes System?
Im Gegensatz zu den Tightly Coupled Systems (Parallel-Systeme) besitzen verteilte Systeme jeweils einen eigenen Prozessor mit einem eigenen Takt und eigenem Hauptspeicher. Sie kommunizieren über stattdessen z.B. über ein lokales Netzwerk oder Telefonleitungen. Sie werden daher auch **Loosely Coupled Systems** oder auch **Distributed Systems** (**Verteilte Systeme**, siehe Kapitel 13) genannt.

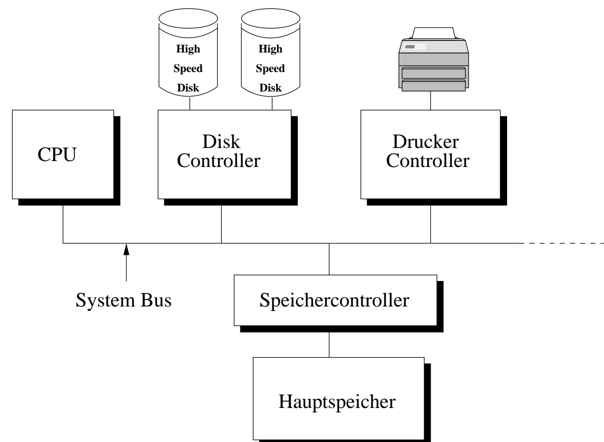


Abbildung 3: Struktur eines Computersystems

2 Computersystem-Komponenten

Frage 2.1: Wie ist ein Computersystem grob aufgebaut?

Ein modernes Computersystem besteht im groben aus den folgenden Komponenten:

- CPU
- Geräte-Controller (**Device Controller**), die z.B. verantwortlich sind für Grafik, Sound, Festplatten, etc.
- (Geteilter) Hauptspeicher

Die CPU und die verschiedenen Controller können dabei nebenläufig arbeiten, wobei sie jedoch um Speichertakte (**Memory Cycles**) konkurrieren. Der Zugriff auf den Hauptspeicher wird von einem Speicher-Controller kontrolliert.

Frage 2.2: Was passiert beim booten eines Betriebssystems?

Wenn ein Computer angeschaltet oder rebootet wird, startet ein simples Programm, welches **Bootstrap Program** genannt wird. Dieses Programm initialisiert alle Aspekte des Systems, wie z.B. die CPU-Register und die Geräte-Controller und lädt dann den Kernel des Betriebssystems. Nachdem der Kernel sich initialisiert hat, wird dann der allererste Prozess (z.B. „init“) gestartet und auf weitere Ereignisse (z.B. in Form von Interrupten) gewartet.

Frage 2.3: Auf welche Arten kann ein Interrupt ausgeführt werden?

Ein Interrupt kann per Hardware oder per Software ausgeführt werden. Wird ein Interrupt per Software ausgeführt, so spricht man von einem **System Call** oder auch **Monitor Call**.

Frage 2.4: Welche Ereignisse können einen Interrupt auslösen. Nenne einige.

- Die Beendigung einer I/O-Operation.

- Division durch Null.
- Unberechtigter Speicherzugriff.
- „Beantragung“ eines Betriebssystem-Dienstes.

Frage 2.5: Wie läuft ein Interrupt in der Regel ab und wie funktioniert er?
Wenn ein Interrupt die CPU erreicht, stoppt sie, was sie gerade macht, und transferiert die Kontrolle an eine bestimmte Stelle im System. Diese Stelle enthält in den meisten Fällen die Adresse, wo sich die Interrupt-Service-Routine befindet, welche ausgeführt wird. Nach Beendigung fährt die CPU an der Stelle fort, an der sie unterbrochen wurde.

Frage 2.6: Auf welche Arten kann ein Interrupt behandelt werden?

Eine allgemeine Routine: Bei dieser Methode wird eine Routine aufgerufen, welche die Interruptinformationen betrachtet und dementsprechend eine interruptspezifische Routine aufruft.

Interruptvektor: Da Interrupte schnell behandelt werden müssen, wird meist dieses Verfahren benutzt. Hier wird ein Array im unteren Speicherbereich (z.B. die ersten hundert Adressen) benutzt, in welchem sich die Adressen der Interrupt-Service-Routinen für die einzelnen Geräte befinden. Diese Array wird über eindeutige Gerätenummern indiziert. Jedes Gerät, welches einen Interrupt auslösen kann, bekommt eine solche Gerätenummer. Diese Methode wird z.B. von so verschiedenen Betriebssystemen wie MS-DOS oder Unix benutzt.

Frage 2.7: Was muss bei der Bearbeitung von Interrupten beachtet werden?
Wird ein Interrupt bearbeitet, so muss die gerade unterbrochenen Instruktion gespeichert werden, damit nach dem Interrupt mit dieser weitergemacht werden kann. Diese kann an einer festgelegten Speicherstelle abgelegt werden, oder aber an einer durch die Gerätenummer indizierten Stelle oder dem System-Stack. Verändert die Interrupt-Service-Routine den Prozessor-Zustand (z.B. einige Register), so muss dieser Zustand vorher explizit gespeichert und nach der Bearbeitung des Interrupt wieder hergestellt werden.

Frage 2.8: Wie ist ein typischer Geräte-Controller aufgebaut?
Ein typischer Controller besitzt einige spezielle Register, sowie einen lokalen Pufferspeicher (**Local Buffer Storage**). Er ist dafür verantwortlich, Daten zwischen dem Puffer und dem Gerät hin- und herzutransferieren. Die Größe des Puffers variiert dabei von Controller zu Controller. Der Puffer eines Festplatten-Controllers hat z.B. die Größe der kleinsten adressierbaren Einheit (ein **Sektor** oder 512 Bytes) oder ein Vielfaches davon.

Frage 2.9: Wie wird eine Ein-/Ausgabe-Operation veranlasst?
Um eine I/O-Operation zu veranlassen, lädt die CPU die jeweiligen Register des Geräte-Controllers, welcher seinerseits den Inhalt dieser Register analysiert

und die entsprechende angeforderte Operation durchführt (z.B. das Laden von Daten). Ist der Transfer komplett löst der Controller einen Interrupt aus.

Frage 2.10: Was ist der Unterschied zwischen „synchroner" und „asynchroner" Ein-/Ausgabe (I/O)?

Bei synchroner I/O wird die I/O gestartet und die Kontrolle erst wieder an den Prozess zurück gegeben, wenn diese beendet ist. Bei asynchroner I/O wird die Kontrolle nach dem Start sofort an den Prozess zurück gegeben, so dass während der I/O noch andere Systemoperationen möglich sind.

Frage 2.11: Welche Möglichkeiten gibt es auf die Beendigung von I/O zu warten?

Entweder besitzt der Prozessor eine sogenannte `wait`-Instruktion, die die CPU so lange anhält bis ein Interrupt ausgelöst wird oder man benutzt einen **wait loop**, der wie folgt aussieht:

```
Loop: jmp Loop
```

Ein solcher Loop muss eventuell noch Geräte abfragen (**pollen**), welche keine Interrupts beherrschen. Dabei wird zum Beispiel nachgesehen ob ein Flag in einem Register gesetzt ist.

Frage 2.12: Was sind die Vor- bzw. Nachteile von synchroner I/O?

Ein Vorteil synchroner I/O ist, dass man immer genau weiß, welches Gerät den Interrupt auslöst. Der Nachteil dieses Vorgehens ist, dass keine Möglichkeit nebenläufiger I/O besteht.

Frage 2.13: Was funktioniert asynchrone I/O?

Bei asynchroner I/O wird, nachdem die I/O angestoßen wurde, anderer Betriebssystem- oder Userprogrammcode ausgeführt. Da I/O nun auch nebenläufig, das heißt parallel, erfolgen kann, wird eine Tabelle benötigt, in der der Status eines jeden Gerätes festgehalten wird. Diese Tabelle heißt **Device-Status Table**. Da es möglich ist, dass verschiedene Prozesse auf dasselbe Gerät zugreifen wollen, muss eine Warteschlange (**Wait Queue**) vorhanden sein.

Tritt ein Interrupt auf, wird zuerst nachgeschaut, welches Gerät den Interrupt verursacht hat. Über diese Information wird in den Device-Status Table indiziert und die entsprechenden Einträge aktualisiert.

Der Hauptvorteil asynchroner I/O ist erhöhte Systemeffizienz, da I/O im Vergleich zur CPU-Geschwindigkeit sehr langsam ist und die CPU auf diese Art und Weise andere nützliche Dinge tun kann.

Frage 2.14: Was versteht man unter DMA?

Wird eine Eingabe von einem 9600-Baud-Terminal eingelesen, so bleibt der CPU nach der Bearbeitung des Interruptes noch genug Zeit, um andere Aufgaben zu erfüllen. Festplatten oder Netzwerkadapter arbeiten jedoch sehr schnell (fast so schnell wie der Hauptspeicher selbst), so dass der CPU in diesem Fall nicht mehr sehr viel Zeit für andere sinnvolle Dinge bliebe, wenn für jedes Byte ein Interrupt ausgelöst würde. Um dieses Problem zu lösen wurde **DMA (Direct Memory Access)** eingeführt.

DMA funktioniert so, dass dem Geräte-Controller gesagt wird, welche Daten wohin in den Hauptspeicher bewegt werden sollen und der Controller dies

dann selbständig macht, ohne Intervention durch die CPU. Dabei wird nur für jeden transferierten Block ein Interrupt ausgelöst anstatt für jedes Byte. Bei der Übertragung der Daten „stiehlt“ der DMA-Controller der CPU jedoch Speicherzyklen (**Memory Cycles**), was dazu führt, dass das Ausführen von Instruktionen durch die CPU etwas abgebremst werden kann.

Frage 2.15: Wofür wird Schutz (Protection) benötigt?

In früheren Systemen, in denen jeweils immer nur ein Programm ausgeführt wurde, konnte ein Fehler des Programms immer nur dieses Programm selbst beeinflussen. In einer Multiprogramming-Umgebung kann es jedoch vorkommen, dass der Fehler eines Programms ein anderes Programm oder dessen Daten beeinflußt. Daher wird ein Mechanismus benötigt, der dies verhindert.

Viele Programmierfehler können direkt durch die Hardware erkannt werden. Wird ein solcher Fehler erkannt, so wird ein Trap zum Betriebssystem ausgelöst, welches sich um den Fehler kümmert. Meist wird das Programm abnormal beendet und eventuell noch der Speicher gedumpt (z.B. Core-Datei).

Frage 2.16: Welche Arten von Schutz werden benötigt, damit ein Betriebssystem reibungslos funktionieren kann?

- Dual-Mode Operation
- I/O Protection
- Memory Protection
- CPU Protection

Frage 2.17: Was versteht man unter „Dual-Mode Operation“?

Dual-Mode Operation ist ein grundlegender Mechanismus, um ein Computersystem gegenüber boshaften oder maliziösen Programmen zu schützen. Dabei wird zwischen zwei Ausführungsmodi unterschieden:

- **User Mode**
- **Monitor Mode** (oder auch **Supervisor Mode, System Mode, Privileged Mode**)

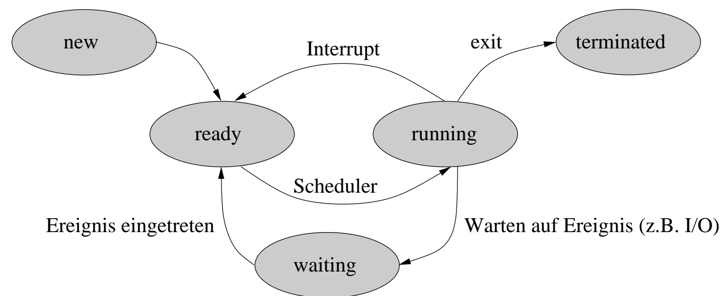


Abbildung 4: Die fünf Zustände, in denen sich ein Prozess befinden kann

3 Prozesse

3.1 Konzept eines Prozesses

Frage 3.1: Was versteht man unter einem Prozess?
Ein Prozess ist ein Programm, welches sich in der Ausführung befindet. Es ist dabei wichtig, zwischen einem Programm, welches eine *passive* Entität ist, und einem Prozess, welcher eine *aktive* Entität ist, zu unterscheiden. Ein Prozess beinhaltet unter anderem:

- Den Programm-Code selbst (**Text Section**).
- Programmzähler (**Program Counter**).
- Inhalt der Register des Prozessors.
- Den Prozess-Stack, der temporäre Daten (Methodenparameter, etc.) enthält.
- Globale Variablen (**Data Section**).

Frage 3.2: Prozesse ändern im Laufe der Zeit ihren Status (Zustand). Welche kennst du?
Ein Prozess kann sich in einem einem dieser fünf Zustände befinden:

New: Der Prozess wird gerade erzeugt.

Running: Der Prozess wird gerade auf einer CPU ausgeführt.

Waiting: Der Prozess wartet auf ein Ereignis (z.B. Beendigung einer Ein-/Ausgabe).

Ready: Der Prozess ist bereit, ausgeführt zu werden, und wartet darauf durch einen Prozessor ausgeführt zu werden.

Terminated: Der Prozess hat seine Ausführung abgeschlossen (oder sie wurde von einem anderen Prozess abgeschlossen).

Pointer	Prozess–zustand
Prozessnummer	
Programmzähler	
Register	
Memory limits	
Liste offener Dateien	
etc.	

Abbildung 5: Inhalt des Process Control Block (PCB)

Abbildung 4 zeigt die fünf Zustände und einige Ereignisse, welche den Zustand eines Prozesses ändern können. Hierbei ist wichtig zu beachten, dass auf einer CPU immer nur ein Prozess im Zustand „running“ sein kann.

Frage 3.3: Wodurch wird ein Prozess im Betriebssystem repräsentiert?
Ein Prozess wird im Betriebssystem, wie in Abbildung 5 dargestellt, durch einen **Process Control Block** oder kurz **PCB** repräsentiert. Der PCB beinhaltet unter anderem:

Prozess-Status: Dies kann sein: new, running, waiting, ready, terminated.

Programmzähler: Beinhaltet den Programm-Zähler.

CPU-Register: Variiert von Architektur zu Architektur in der Anzahl der Register, etc. Diese Information muss bei einem Interrupt unter anderem gespeichert werden.

CPU-Scheduling-Information: Beinhaltet z.B. die Priorität eines Prozesses, aber auch Zeiger auf Scheduling-Warteschlangen, in denen sich der Prozess befindet.

Memory-Management-Information: Abhängig vom Speicher-System des Betriebssystems.

Accounting-Information: Der Betrag an CPU- bzw. Realzeit, die der Prozess benutzt hat, Zeitlimits und ähnliches.

I/O-Status-Information: Liste von geöffneten Dateien und verwendeten Geräten, etc.

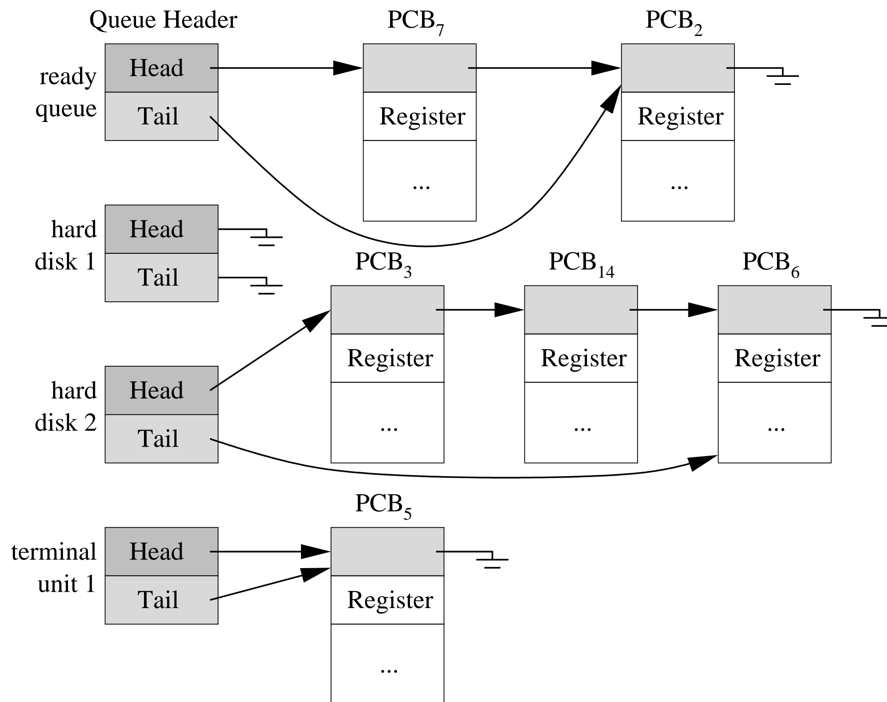


Abbildung 6: Ready und I/O-Queue

3.2 Scheduling von Prozessen

Frage 3.4: Welche Scheduling-Queues werden für Prozesse benötigt?
Die folgenden Queues lassen sich in einem System finden:

Job Queue: In dieser Queue befinden sich alle Prozesse, die sich im System befinden.

Ready Queue: In der Ready Queue befinden sich alle Prozesse, die bereit sind ausgeführt zu werden. Diese Queue ist oft als lineare Liste implementiert. Daher beinhaltet der PCB oft noch einen Zeiger auf seinen Nachfolger in dieser Queue.

Device Queue: Da es vorkommen kann, dass verschiedene Prozesse auf dasselbe Gerät zugreifen wollen, das Gerät jedoch gerade für einen anderen Prozess beschäftigt ist, werden wartende Prozesse in der Geräte-Warteschlange verwaltet.

Frage 3.5: Wie bewegen sich Prozesse in der Regel von Queue zu Queue?
In der Regel wird ein Prozess irgendwann von der Job Queue in die Ready Queue bewegt, so dass er auf der CPU ausgeführt werden kann. Bei der Ausführung auf der CPU kann es zu folgenden Ereignissen kommen:

- Der Prozess fragt nach einer Ein- oder Ausgabe und wird in die I/O-Queue bewegt.

- Der Prozess erzeugt einen Unterprozess und wartet auf dessen Terminierung.
- Der Prozess wird durch einen Interrupt von der CPU genommen und wieder in die Ready Queue gepackt.

Terminiert ein Prozess, so wird er aus allen Queues entfernt und alle Resourcen deallokiert. Abbildung 6 zeigt die Ready Queue und einige Gerätequeues.

Frage 3.6: Was ist ein Scheduler? Was für Arten von Schedulern gibt es und was sind ihre Aufgaben?

Die Entscheidung, ob eine Prozess von einer Queue in die nächste wandert, wird von sogenannten Schedulern getroffen. Dabei gibt es die folgenden Arten von Schedulern:

Long-Term Scheduler: Der Long-Term Scheduler entscheidet, welche Prozesse aus dem Job Pool (diese Prozesse befinden sich teilweise z.B. auf einer Festplatte) in den Hauptspeicher geladen werden, damit sie ausgeführt werden. Einige Systeme besitzen keinen oder nur einen minimalen Long-Term Scheduler. So wird z.B. bei UNIX jeder Prozess in den Hauptspeicher geladen.

Short-Term Scheduler: Der Short-Term Scheduler entscheidet, welcher Prozess aus der Ready Queue als nächstes auf der CPU ausgeführt wird.

Medium-Term Scheduler: Die Hauptidee beim Medium-Term Scheduler ist es, dass es teilweise vorteilhaft sein kann, Prozesses aus dem Hauptspeicher herauszunehmen, z.B. um einen besseren Prozess-Mix zu erreichen.

Frage 3.7: Was versteht man unter einem Prozess-Mix?

Man kann grob zwischen zwei verschiedenen Arten von Prozessen unterscheiden. Prozesse die **I/O-gebunden (I/O bound)** sind und Prozesse die **CPU-gebunden (CPU bound)** sind. Ein I/O-gebundener Prozess verbringt die meiste Zeit mit Ein-/Ausgabe, während ein CPU-gebundener Prozess fast nur Berechnungen durchführt und nur selten Ein-/Ausgabe tätigt. Ein wichtiges Ziel des Long-Term Schedulers ist es, eine ausgewogene Mischung aus I/O-gebundenen und CPU-gebundenen Prozessen zu finden, damit der Rechner optimal genutzt wird.

Frage 3.8: Was ist ein Context Switch?

Ein Wechsel der CPU auf einen anderen Prozess wird **Context Switch** genannt. Dabei muss der Zustand des alten Prozesses gespeichert werden und der des neuen geladen werden. Der Kontext eines Prozesses ist durch den PCB repräsentiert. Ein Context Switch ist purer Overhead, da während dessen keine nützliche Arbeit verrichtet wird. Die Dauer eines Context Switch kann architekturabhängig sein und unterschiedlich lange dauern (zwischen 1 und 1000 Mikrosekunden).

3.3 Operationen auf Prozessen

Frage 3.9: Wie werden unter Unix Prozesse erzeugt?

Prozesse werden unter UNIX mit dem `fork` System Call erzeugt. Der Unterprozess bzw. Kindprozess (**Child Process**) besteht dabei aus einer Kopie des Adressraums des erzeugenden Prozesses (**Parent Process**). Beide Prozesse fahren nach dem `fork`-Aufruf fort, mit dem Unterschied, dass der Aufruf im Elternprozess die Prozess-ID (**process id** oder **pid**) des Kindprozesses zurück gibt, während der Aufruf im Kindprozess den Wert 0 zurück gibt. Anhand dessen können die Prozesse unterschieden werden und unterschiedlich fortgefahren werden.

Typischerweise wird der `execlp` System Call direkt nach dem `fork` ausgeführt, welcher ein Programm in den neuen Prozess lädt und dieses ausführt.

Frage 3.10: Wie wird ein Prozess unter Unix terminiert?

Ein Prozess wird unter UNIX mit Hilfe des `exit` System Calls terminiert. Dies geschieht entweder, nachdem die letzte Instruktion ausgeführt wurde oder durch einen anderen Prozess, der üblicherweise der Elternprozess des zu terminierenden Prozesse sein muss. Wird ein Prozess terminiert, so werden alle von ihm allokierten Resourcen wieder freigegeben.

Frage 3.11: Was versteht man unter Cascading Termination?

Bei einigen Systemen werden automatisch auch alle Kindprozesse terminiert, wenn der Elternprozess terminiert. Dieses Phänomen wird **Cascading Termination** genannt. Unter UNIX und LINUX wird der **init**-Prozess der neue Elter von Kindprozessen, dessen ursprünglicher Elter terminiert.

3.4 Kooperierende Prozesse

Frage 3.12: Was ist ein unabhängiger und was ein kooperativer Prozess?

Ein unabhängiger Prozess ist ein Prozess, der keine anderen Prozesse beeinflussen kann und auch selbst nicht durch andere Prozesse beeinflusst werden kann. So ist z.B. jeder Prozess, der keine Daten mit anderen Prozessen teilt unabhängig. Ein kooperativer Prozess kann andere Prozess beeinflussen oder durch andere Prozesse beeinflusst werden.

Frage 3.13: Welches Problem illustriert das Konzept kooperierender Prozess besonders gut?

Das **Producer-Consumer Problem** illustriert kooperierende Prozesse besonders gut. Ein Produzent (**Producer**) produziert Informationen, welche von einem Konsumenten (**Consumer**) verbraucht werden. Die Informationen werden dabei in einem Puffer abgelegt und auch von dort abgeholt. Beide Instanzen müssen also Zugriff auf diesen Puffer haben. Beim **unbounded-buffer** Producer-Consumer Problem kann der Produzent soviel produzieren, wie er will, während beim **bounded-buffer** Producer-Consumer Problem der Puffer eine feste Größe hat.

Produzent und Konsument müssen sich synchronisieren, damit der Konsument nicht versucht, eine Information zu konsumieren, welche noch nicht

produziert wurde und damit der Produzent beim bounded-buffer Producer-Comsumer Problem nicht eine Information produziert, wenn der Puffer schon voll ist.

3.5 Interprozess-Kommunikation

Frage 3.14: Welche Möglichkeit gibt neben geteiltem Speicher („shared memory“) noch, um Kommunikation zwischen Prozessen zu ermöglichen?
Dazu kann die sogenannte **Interprocess-Communication (IPC)** Einrichtung benutzt werden. Diese wird vom Betriebssystem zur Verfügung gestellt. Prozesse können mittels dieser kommunizieren, sowie sich synchronieren, ohne dass sie ihren Adressraum teilen müssen. Dies erweist sich in verteilten Systemen als äußerst nützlich, da hier Prozesse miteinander kommunizieren können, welche auf Rechnern laufen, die weit voneinander entfernt sind, also sich keinen physikalischen Speicher teilen. IPC lässt sich am Besten mittels **Message Passing**-Systemen realisieren.

Frage 3.15: Beschreibe allgemein ein IPC!
IPC besitzt immer mindestens zwei Operationen. Eine zum Versenden von Nachrichten, `send(message)`, und eine zum Empfangen von Nachrichten, `receive(message)`. Die Größe der Nachricht kann dabei fest oder variabel sein. Ein IPC, welches Nachrichten fester Grösse verschickt, ist leichter zu implementieren, aber schwieriger zu benutzen, als ein IPC, welches eine variable Nachrichtengröße erlaubt. Möchten nun zwei Prozesse P und Q miteinander kommunizieren, so müssen sie sich gegenseitig Nachrichten senden und empfangen, es muss ein **Communication Link** zwischen ihnen bestehen. Bei der Implementierung eines IPC, müssen die folgenden Design-Fragen berücksichtigt werden:

- Direkt vs. indirekte Kommunikation
- Symmetrische vs. asymmetrische Kommunikation
- Automatisches vs. explizites Puffern
- Kopie vs. Referenz versenden
- Feste vs. variabler Größe der Nachtrichten

Frage 3.16: Wie kann eine direkte Kommunikation mittels Message Passing Systems realisiert werden? Wie eine indirekte?
Direkte Kommunikation bedeutet, dass ein Prozess, welcher mit einem anderen Prozess kommunizieren möchte, diesen explizit benennen muss. Die `send` und `receive` Befehle besitzen folgenden Aufbau:

send(P, message): Sende eine Nachricht an Prozess P.

receive(Q, message): Empfange eine Nachricht von Prozess Q.

Die oben angegebenen Befehle beziehen sich auf eine symmetrische Kommunikation, da der Empfänger die Nachricht unter Angabe des Senders empfängt. Eine direkte, asymmetrische Kommunikation würde einen receive-Befehl vorsehen, in welchem der Sender einer Nachricht nicht explizit angegeben werden muss. Dies kann die Form `receive(id, message)` haben, wobei das System in die Variable `id` jeweils die Prozess-ID des Senders hineinschreiben würde.

Wird ein IPC mittels der obigen Befehle realisiert, so besitzt der Communication Link folgende Eigenschaften:

1. Ein Link wird automatisch zwischen jedem Paar von Prozessen hergestellt, welche miteinander kommunizieren möchten. Die Prozesse müssen lediglich ihre Identität kennen, um miteinander kommunizieren zu können.
2. Ein Link verbindet immer genau zwei Prozesse.
3. Es existiert immer genau ein Link zwischen zwei Prozessen.

Wird IPC mittels indirekter Kommunikation realisiert, so geschieht die Kommunikation über eine **Mailbox** (auch **Port** genannt). Alle Nachrichten werden an diese geschickt und auch von ihr empfangen. Jede Mailbox verfügt über eine eindeutige Identifikationsnummer. Ein Prozess kann somit über verschiedene Mailboxen mit beliebig vielen anderen Prozessen kommunizieren, aber zwei Prozesse können nur kommunizeiren, wenn sie sich eine Mailbox „teilen“. Die `send` und `receive` Befehle sind wie folgt definiert:

send(A, message): Sende Nachricht an Mailbox A.

receive(A, message): Empfange eine Nachricht von Mailbox A.

Damit ergeben sich folgende Eigenschaften:

1. Ein Link zwischen zwei Prozessen besteht nur, wenn diese sich eine Mailbox teilen.
2. Ein Link kann mehr als zwei Prozessen zugeordnet sein.
3. Zwischen zwei Prozessen kann mehr als ein Link existieren, wobei jeder Link einer Mailbox zugeordnet werden kann.

Wie können einem Link mehr als zwei Prozesse zugeordnet sein? Dies ist immer dann der Fall, wenn sich mehr als zwei Prozesse eine Mailbox teilen. Dabei kann es jedoch passieren, dass der Empfänger einer Nachricht nicht mehr eindeutig zuordbar ist. Ein Beispiel: Die Prozesse P_1, P_2 und P_3 teilen sich eine Mailbox A. P_1 schickt eine Nachricht an A, während P_2 und P_3 ein `receive(A, message)` versenden. Welcher Prozess wird die von P_1 gesendete Nachricht empfangen? Die Antwort auf diese Frage hängt davon ab, welches der folgenden drei Schemata gewählet wird:

1. Eine Link kann nur zwischen maximal zwei Prozessen bestehen.
2. Zu jedem Zeitpunkt darf nur ein Prozess eine `receive` Anforderung abschicken.

3. Das Betriebssystem entscheidet beliebig, welcher Prozess (im obigen Beispiel P_2 oder P_3) die Nachricht erhält. Es kann dann eine Benachrichtigung über den Empfänger an den Sender geschickt werden.

Wem gehört nun eine Mailbox? Es gibt zwei Möglichkeiten. Sie kann entweder dem Betriebssystem gehören oder einem Prozess. In diesem Fall ist sie ein Teil seines Adressraums. Gehört sie einem Prozess, dann kann zwischen Besitzer und Benutzer unterschieden werden. Der Besitzer kann über diese Mailbox nur empfangen, ein Benutzer kann ausschließlich an die Mailbox senden. Da jede Mailbox einen eindeutigen Besitzer hat, besteht keine Konfusion, es ist immer klar, wer eine an diese Mailbox gesendete Nachricht empfängt. Wird der Prozess terminiert, so müssen alle Prozesse, welche an diese Mailbox gesendet haben, darüber informiert werden, da mit dem Prozess auch die Mailbox gelöscht wird.

Werden die Mailboxen hingegen vom Betriebssystem verwaltet und gehören somit diesem, so muss es den Prozessen Funktionen zur Verfügung stellen, mittels welchen sie eine neue Mailbox öffnen, über diese kommunizieren und sie schliesslich löschen können. Erstellt ein Prozess eine neue Mailbox, so wird er als Besitzer eingetragen. Nur er kann zu Beginn über über diese Mailbox Nachrichten empfangen. Diese Empfangsrechte können zur Laufzeit vom Prozess erweitert werden, wobei man dann in die oben beschriebene Situation geraten kann.

Frage 3.17: Nenne Vor- und Nachteile direkter und indirekter Kommunikation!

Direkte Kommunikation kommt im Gegensatz zu indirekter Kommunikation mit relativ wenig Overhead aus. Es muss nicht erst eine Mailbox erschaffen werden und auch die Kontrollmechanismen entfallen. Allerdings kommt es zu Problemen, wenn sich aus irgendeinem Grund der Name eines Prozesses ändert. Dann müssen alle Instanzen des Namens gefunden und entsprechend verändert werden. Bei modularen Programmen kann dies zu Problemen führen. Diese Probleme werden mittels indirekter Kommunikation umgangen, allerdings führt dies zum bereits erwähnten Overhead.

Frage 3.18: Wie kann die Kommunikation von Prozessen synchronisiert werden?

Message Passing kann synchron oder asynchron realisiert werden. Synonym können auch die Begriffe *blocking* (synchron) und *non-blocking* (asynchron) benutzt werden.

Blocking Send: Der sendende Prozess ist solange blockiert, bis die Nachricht vom Zielprozess oder der entsprechenden Mailbox empfangen wurde.

Non-Blocking Send: Der sendende Prozess sendet die Nachricht und fährt seinen Berechnungen fort.

Blocking Receive: Der Empfänger blockt solange, bis eine Nachricht eingetroffen ist.

Non-Blocking Receive: Der Empfänger empfängt entweder eine gültige Nachricht oder *null*, falls keine Nachricht gesendet wurde.

Blocken sowohl Sender als auch Empfänger, so bezeichnet man dies als **Rendezvous**.

Frage 3.19: Was versteht man unter Buffering und welche verschiedenen Möglichkeiten des Bufferings sind Ihnen bekannt?

Egal ob die Kommunikation direkt oder indirekt ist, Nachrichten, welche zwischen kommunizierenden Prozessen ausgetauscht werden, befinden sich in einer temporären Queue. Eine solche Queue kann auf drei verschiedene Arten implementiert werden:

Zero Capacity: Die Queue hat eine maximale Länge von 0, der Link kann also keine wartenden Nachrichten speichern. In diesem Fall muss der Sender solange blocken, bis der Empfänger die Nachricht abgerufen hat. Es liegt ein Rendezvous vor.

Bounded Capacity: Die Queue hat eine Länge von n. Befinden sich weniger als n Nachrichten in der Queue, wenn ein Sender eine Nachricht schickt, so kann er nach dem Versenden der Nachricht weiterrechnen. Ist der Link hingegen voll, so muss der Sender solange blocken, bis der Empfänger eine Nachricht per `receive` abgeholt hat, also wieder Platz in der Queue ist.

Unbounded Capacity: Es können beliebig viele Nachrichten in der Queue gespeichert werden, der Sender muss niemals blocken.

Der erste Fall wird als Message Passing ohne Buffering, die beiden anderen Fälle als automatisches Buffering bezeichnet.

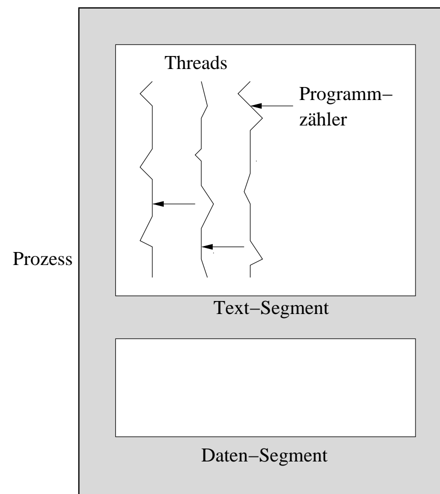


Abbildung 7: Mehrere Threads in einem Prozess

4 Threads

Frage 4.1: Was ist ein Thread?

Herkömmliche Prozesse besitzen nur einen „Code-Strang“, der sequentiell abgearbeitet wird. In modernen Betriebssystemen ist es jedoch möglich, dass ein Prozess mehrere „Code-Stränge“ besitzt und so mehr als eine Aufgabe auf einmal bearbeiten kann bzw. wenn eine blockende Operation durchgeführt wird, die CPU für eine andere Aufgabe nutzen kann. Abbildung 7 vedeutlicht dies.

Ein **Thread** besteht grob aus einer Thread-ID, einem Programmzähler, einem Register-Set und einem Stack und ist eine grundlegende Einheit der CPU-Nutzung (*„A thread [...] is a basic unit of CPU utilization...“*). Er teilt sich mit anderen zu einem Prozess gehörigen Threads die Code-Sektion, die Daten-Sektion, sowie andere Betriebssystem-Resourcen wie z.B. geöffnete Dateien und Signale.

Frage 4.2: Gebe ein Beispiel für eine Applikation, wo Threads sinnvoll sind!

In einer Textverarbeitung kann z.B. ein Thread benutzt werden, um Grafiken zu zeichnen, während ein anderer Benutzereingaben verarbeitet und wieder ein anderer im Hintergrund eine Rechtschreibprüfung vornimmt.

Bei Servern kann die Nutzung von Threads ebenfalls sehr sinnvoll sein, da verschiedene Clients sonst unter Umständen sehr lange auf Antworten warten müssen, wenn diese nacheinander bearbeitet werden.

Frage 4.3: Welche Vorteile bringen Threads?

Die Vorteile des Programmierens mit Threads kann man in vier Kategorien einteilen:

Responsiveness: Durch Multithreading kann die Ansprechbarkeit (**responsiveness**) eines Programms erhöht werden, da einzelne Programmteile weiterlaufen können, auch wenn andere Teile blockende oder lang andauernde Operationen durchführen.

Teilen von Resourcen (Resource Sharing): Threads teilen sich die Resourcen des Prozesses, zu dem sie gehören.

Ökonomie (Economy): Neue Prozesse zu erzeugen, kann sehr teuer werden, da erst Speicher und andere Resourcen allokiert werden müssen. Da sich Threads die Resourcen ihres Prozesses teilen, ist es sparsamer, Threads zu erzeugen. Weiterhin geht ein Context Switch bei Threads ebenfalls schneller vonstatten als ein Context Switch zwischen verschiedenen Prozessen.

Nutzung von Mehrprozessor-Architekturen: In einem Multiprozessor-System können die Threads auf verschiedene Prozessoren verteilt und ausgeführt werden.

Frage 4.4: Welche zwei Arten von Threads kann man unterscheiden?
Es kann zwischen **User Threads** und **Kernel Threads** unterschieden werden. In ersterem Fall wird Thread-Support auf der Benutzerebene gegeben, im zweiteren auf der Kernelebene.

Frage 4.5: Was sind User Threads? Was sind ihre Vor- und Nachteile?
User Threads werden oberhalb des Kernels durch eine Thread-Bibliothek implementiert, d.h. auf Benutzerebene. Da für die Erzeugung, das Scheduling und das Management der Threads keine Kernelunterstützung benötigt wird, sind User Threads in der Regel recht schnell zu erzeugen und zu managen. Sie haben jedoch auch Nachteile: Ist der Kernel **single threaded**, wird jede blockierende Operation eines User Threads den ganzen Prozess blockieren, auch wenn andere Threads laufen könnten. Beispiele für User Thread Bibliotheken sind Posix Pthreads, Mach C-Threads und Solaris Threads.

Frage 4.6: Was sind Kernel Threads? Was sind ihre Vor- und Nachteile?
Kernel Threads werden direkt durch das Betriebssystem angeboten, d.h. die Erzeugung von Threads, ihr Scheduling und Management wird direkt im **Kernel Space** durch den Kernel ausgeführt. In der Regel sind sie daher langsamer zu erzeugen und zu managen, bieten jedoch den Vorteil, dass wenn ein Thread eine blockende Operation durchführt, ein anderer Thread geschedult werden kann. Desweiteren können die Threads in Multiprozessor-Systemen auf verschiedene Prozessoren verteilt werden. Betriebssysteme, die Kernel Threads unterstützen sind z.B. Windows NT, Solaris und Digital Unix.

Frage 4.7: Welche Multithreading-Modelle kennst du?
Viele Systeme bieten Support für User Threads und Kernel Threads, was zu verschiedenen Multithreading-Modellen führt. Man kann unterscheiden zwischen:

Many-to-One Modell: In diesem Modell werden mehrere User Threads auf einen Kernel Thread abgebildet. Hier werden die Threads im **User Space** verwaltet, was recht effizient ist, jedoch blockiert ein blockierender System Call den ganzen Prozess. Betriebssysteme, die keinen Support für Kernel Threads haben, benutzen dieses Modell.

One-to-One Modell: In diesem Modell wird jeder User Thread auf einen Kernel Thread abgebildet. Diese Modell bietet mehr Nebenläufigkeit als das Many-to-One Modell, da einfach ein anderer Thread ausgeführt wird, wenn ein Thread einen blockierenden System Call macht. Desweiteren können die Threads in diesem Modell auf mehrere Prozessoren verteilt werden. Da die Erzeugung von Kernel Threads jedoch mit einem gewissen Overhead verbunden ist, beschränken viele Systeme die maximale Anzahl der Threads, die erzeugt werden dürfen.

Many-to-Many Modell: Dieses Modell multiplext mehrere User Threads auf weniger oder gleich viele Kernel Threads. Es stellt quasi einen Kompromiss zwischen dem Many-to-One Modell, wo der Benutzer so viele Threads erzeugen kann, wie er möchte, und dem One-to-One Modell, in dem wirkliche Nebenläufigkeit erreicht wird.

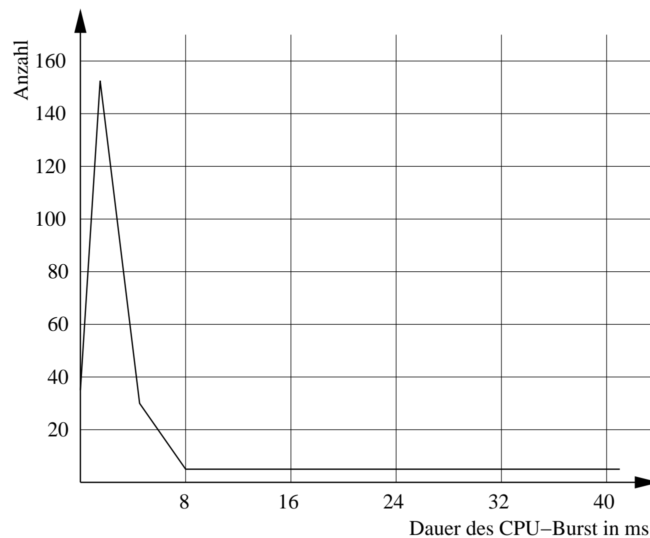


Abbildung 8: Histogramm von CPU Bursts

5 CPU Scheduling

5.1 Einleitung

Frage 5.1: Noch einmal zur Erinnerung: Warum wird CPU Scheduling gemacht?

CPU Scheduling ist die Grundlage des Multiprogramming, dessen Ziel es ist, dass immer mindestens ein Prozess von der CPU ausgeführt wird. Auf diese Art und Weise wird das System effizient genutzt. Es befinden sich dazu mehrere Prozesse im Hauptspeicher. Wartet ein Prozess (z.B. auf die Beendigung einer Ein-/Ausgabe) wird die CPU einem anderen Prozess gegeben, bis dieser eventuell irgendwann warten muss oder ein Interrupt auftritt.

Frage 5.2: Von welcher bei Prozessen beobachtbaren Eigenschaft hängt CPU Scheduling ab?

Prozesse durchlaufen einen Kreislauf von Ausführung auf der CPU und Warten auf I/O. Ein Prozess startet mit einem **CPU Burst**, gefolgt von einem **I/O Burst**, welcher wiederum von einem CPU Burst gefolgt wird, usw. Der letzte CPU Burst endet in der Regel mit einer System Call-Nachfrage, den Prozess zu terminieren.

Frage 5.3: Wie sieht ein Histogramm von CPU Bursts in der Regel aus?

Ein typische Histogramm von CPU Bursts ist in Abbildung 8 dargestellt. Es gibt wenige Bursts, die sehr kurz sind, relativ viele kurze Bursts und dann wieder weniger lange bis sehr lange Bursts.

Frage 5.4: Welche Möglichkeiten gibt es, die Ready Queue zu implementieren? Was sind ihre Einträge?

Die Ready Queue kann auf verschiedene Arten und Weisen implementiert werden und muss nicht zwingend eine **FIFO Queue** sein. Andere Möglichkeiten

sind eine Prioritätenqueue (**Priority Queue**), ein Baum oder eine einfach eine ungeordnete lineare Liste.

Die Einträge der Ready Queue sind in der Regel die PCBs der Prozesse, die bereit sind, auf der CPU ausgeführt zu werden.

Frage 5.5: Was ist der Unterschied zwischen nicht-preemtiven und preemptiven Scheduling?

Man kann vier Bedingungen unterscheiden, nach denen eine CPU Scheduling-Entscheidung notwendig werden kann:

1. Ein Prozess wechselt vom Zustand „running“ in den Zustand „waiting“ (z.B. nach I/O-Anfrage).
2. Ein Prozess wechselt vom Zustand „running“ in den Zustand „ready“ (z.B. durch einen Interrupt).
3. Ein Prozess wechselt vom Zustand „waiting“ in den Zustand „ready“ (z.B. nach Beendigung von I/O).
4. Ein Prozess terminiert.

Bei Bedingung 1 und 4 hat man keine andere Wahl, als einen anderen Prozess zu starten (zumindest wenn man Multiprogramming betreiben möchte). Wohl aber im Fall 2 und 3. Findet Scheduling nur unter Bedingung 1 und 4 statt, so spricht man von nicht-preemptiven Scheduling, ansonsten von preemptiven Scheduling. Dies bedeutet, dass ein Prozess beim nicht-preemptiven Scheduling so lange die CPU besetzt, bis er entweder in den Zustand „waiting“ wechselt oder bis er terminiert. Nicht-preeemptives Scheduling wird zum Beispiel auf bestimmten Hardware-Plattformen verwendet, denen die Technik fehlt, die für preemptives Scheduling nötig ist, wie z.B. Timer.

Windows 3.1 benutzte noch nicht-preemptives Scheduling. Preemptives Scheduling wurde mit Windows 95 eingeführt.

Frage 5.6: Was ist ein „Dispatcher“?

Der **Dispatcher** ist die Komponente, die einem Prozess die Kontrolle über die CPU gibt. Dafür müssen folgende Schritte durchgeführt werden:

- Context Switch
- Wechsel in den User Mode
- An die richtige Stelle des Prozesses springen, damit er korrekt fortgesetzt wird

Da der Dispatcher bei jedem Prozesswechsel aufgerufen wird, muss er sehr schnell sein. Die Zeit, die der Dispatcher benötigt, um einen Prozess zu stoppen und einen anderen zu starten, wird **Dispatch Latency** genannt.

5.2 Scheduling-Kriterien

Frage 5.7: Anhand welcher Kriterien kann man Scheduling-Algorithmen betrachten und analysieren?

Es gibt verschiedene Kriteria, anhand derer Scheduling-Algorithmen betrachtet und analysiert werden können:

CPU-Nutzung (CPU Utilization): Sinn des CPU Scheduling ist es, die CPU so beschäftigt wie möglich zu halten. Die CPU-Nutzung kann zwischen 0 und 100 Prozent liegen. In einem realen System sollte sie zwischen 40 und 90 Prozent liegen.

Durchsatz (Throughput): Der Durchsatz ist die Menge der Prozesse, die während einer Zeiteinheit beendet werden. Für langsame Prozesse kann dies ein Prozess pro Stunde sein, für schnelle Prozesse 10 pro Sekunde.

Turnaround Time: Die Turnaround Time ist die Zeit, die ein Prozess braucht um ausgeführt zu werden (bis zur Beendigung). Es ist die Summe der Zeiten, die gewartet wird, um in den Hauptspeicher zu gelangen, die in der Ready Queue verbracht wird, die sich der Prozess auf der CPU befindet und in denen I/O gemacht wird.

Wartezeit (Waiting Time): Ein CPU Scheduling-Algorithmus beeinträchtigt nicht die Zeit, die ein Prozess für I/O braucht, wohl aber die Zeit, die ein Prozess in der Ready Queue wartet. Die Wartezeit ist daher die Summe der Perioden, die ein Prozess in der Ready Queue wartet.

Antwortzeit (Response Time): Die Antwortzeit ist die Zeit, die ein Prozess in einem interaktiven System braucht, um auf ein Ereignis (z.B. eine Eingabe) zu reagieren.

Hierbei wird angestrebt, die CPU-Nutzung und den Durchsatz zu maximieren, während die Turnaround Time, die Wartezeit und die Antwortzeit minimiert werden soll. Dabei wird oft versucht, den Durchschnitt dieser Werte zu maximieren bzw. minimieren. Experten haben jedoch herausgefunden, dass es z.B. bei der Antwortzeit sinnvoller ist, die Varianz der Antwortzeiten zu minimieren.

5.3 Scheduling Algorithmen

Frage 5.8: Welche Scheduling-Algorithmen kennst du?

- First-Come, First-Served Scheduling (FCFS)
- Shortest-Job-First Scheduling (SJF)
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback-Queue Scheduling

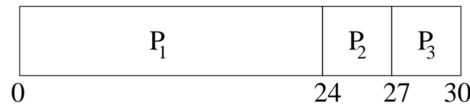


Abbildung 9: FCFS-Scheduling bei der Reihenfolge P_1, P_2 und P_3

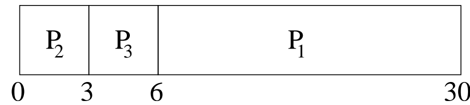


Abbildung 10: FCFS-Scheduling bei der Reihenfolge P_2, P_3 und P_1

Frage 5.9: Wie funktioniert First-Come, First-Served Scheduling?
Beim First-Come, First-Served Scheduling (FCFS) wird dem Prozess zuerst die CPU gegeben, der als erstes danach gefragt hat. Dies wird über eine einfache FIFO-Queue geregelt. Der PCB eines neuen Prozesses wird ans Ende dieser Queue eingefügt. Ist die CPU frei, wird der erste Prozess (am Kopf) aus der Queue herausgenommen und ausgeführt.

Frage 5.10: Was sind die Vor- bzw. Nachteile beim FCFS Scheduling?
Der Vorteil des FCFS Scheduling ist, dass es relativ leicht zu implementieren ist. Leider gibt es auch einige Nachteile. So kann die durchschnittliche Wartezeit beim FCFS Scheduling unter Umständen sehr groß werden. Im allgemeinen ist die durchschnittliche Wartezeit nicht minimal und kann auch sehr variieren. Seien z.B. die folgenden Prozesse mit den dazugehörigen (CPU-)Burst-Zeiten in Millisekunden gegeben:

Prozess	Burst-Zeit
P_1	24
P_2	3
P_3	3

Kommen die Prozesse in der Reihenfolge P_1, P_2 und P_3 zum Zeitpunkt 0 an, so beträgt die durchschnittliche Wartezeit $\frac{0+24+27}{3} = 17$ Millisekunden (Abbildung 9). Wenn sie allerdings in der Reihenfolge P_2, P_3 und P_1 ankommen, so beträgt die durchschnittliche Wartezeit nur noch $\frac{0+3+6}{3} = 3$ Millisekunden (Abbildung 10).

Desweiteren kann es bei dieser Scheduling-Politik zum sogenannten **Convoy-Effekt** kommen. Angenommen unser Prozess-Mix besteht aus einem CPU-gebundenen Prozess und mehreren I/O-gebundenen Prozessen. Nach einiger Zeit kann es zu folgendem Szenario kommen. Der CPU-gebundene Prozess erlangt die CPU und wird eine Weile ausgeführt. Währenddessen wird die I/O der anderen Prozesse beendet. Irgendwann gibt der CPU-gebundene Prozess die CPU wieder frei (z.B. weil er selbst I/O machen muss) und die I/O-gebundenen Prozesse erlangen nacheinander die CPU. Diese werden sich jedoch, aufgrund ihrer Natur, nur kurz auf der CPU befinden. Im Anschluss daran erlangt wieder der CPU-gebundene Prozess die CPU, während die anderen auf den Abschluss ihrer I/O warten. Dies bedeutet, dass fast immer einige Prozesse auf einen großen langsamen Prozess warten müssen.

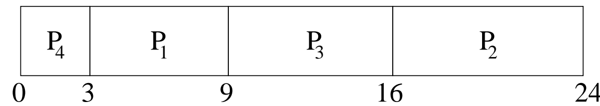


Abbildung 11: SJF-Scheduling

Der FCFS Algorithmus ist nicht-preemptiv, was ihn für Time-Sharing-Systeme unbrauchbar macht. Denn hier ist es wichtig, dass jeder Benutzer in regelmäßigen Zeitabständen einmal die CPU bekommt.

Frage 5.11: Wie funktioniert das Shortest-Job-First Scheduling?

Zuerst einmal sei angemerkt, dass eine angemessenere Bezeichnung **Shortest Next CPU Burst-First** wäre, anstatt **Shortest-Job-First (SJF)**. Bei diesem Verfahren wird mit jedem Prozess seine nächste CPU-Burst-Zeit assoziert und die CPU jeweils dem Prozess mit der kleinsten CPU-Burst-Zeit zugeteilt.

Seien zum Beispiel die folgenden Prozesse mit den entsprechenden Burst-Zeiten gegeben:

Prozess	Burst-Zeit
P_1	6
P_2	8
P_3	7
P_4	3

Dann würden diese in der Reihenfolge P_4, P_1, P_3, P_2 geschedult werden (Abbildung 11). Die durchschnittliche Wartezeit beträgt $\frac{0+3+9+16}{4} = 7$ Millisekunden. Es kann gezeigt werden, dass SJF Scheduling für jede Menge von Prozessen die minimale durchschnittliche Wartezeit liefert.

Frage 5.12: Was passiert bei SJF, wenn zwei Prozesse die gleiche CPU-Burst-Zeit haben?

In diesem Fall wird das Problem per FCFS gelöst, d.h. der Prozess genommen, der zuerst da war.

Frage 5.13: Beweise, dass SJF die minimale durchschnittliche Wartezeit für eine gegebene Menge von Prozessen liefert!

Dies kann anhand der Vertauschung zweier Prozesse gezeigt werden. Beim SJF Scheduling wird ein Prozess mit einer kürzeren Burst-Zeit vor einem Prozess mit einer längeren Burst-Zeit ausgeführt. Eine Vertauschung der beiden erhöht die durchschnittliche Wartezeit des längeren Prozess nicht in dem Maße, wie sie die durchschnittliche Wartezeit des kürzeren Prozess verschlechtert. Ist irgendeine Scheduling-Reihenfolge gegeben, so kann die durchschnittliche Wartezeit also verbessert werden, indem jeweils ein kürzerer Prozess vor einem längeren Prozess ausgeführt wird. Dies führt auf natürliche Art und Weise zum SJF Scheduling.

Frage 5.14: Welche Probleme gibt es bei SJF? Wie können sie gelöst werden?

Ein Problem bei SJF ist, dass die Länge des nächsten CPU-Burst nicht bekannt ist. Daher wird versucht, SJF mit bestimmten Verfahren *anzunähern*. Wenn

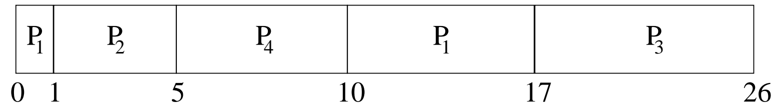


Abbildung 12: Preemptives SJF Scheduling

man die Länge des nächsten CPU-Burst nicht weiß, so kann man zumindest versuchen, sie vorherzusagen. Dabei wird davon ausgegangen, dass der nächste Burst in etwa die gleiche Länge hat, wie der vorherige.

Die Länge des nächsten Burst wird dabei als **exponentieller Durchschnitt** der gemessenen Längen der vorherigen Bursts vorhergesagt. Sei t_n die Länge des n-ten CPU-Burst und τ_{n+1} sei die nächste vorhergesagte Länge. Dann definieren wir für $\alpha, 0 \leq \alpha \leq 1$:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Über den Parameter α kann die Gewichtung der kurz zurückliegenden und länger zurückliegenden Geschichte variiert werden. Setzen wir zum Beispiel $\alpha = 1$, so ist $\tau_{n+1} = t_n$, d.h. uns interessiert nur der zuletzt gemessene Wert. Wird $\alpha = 0$ gesetzt, gilt $\tau_{n+1} = \tau$, d.h. es werden konstante Verhältnisse angenommen. τ_0 wird in der Regel auf eine Konstante gesetzt oder auf einen systemweiten Durchschnitt. In der Regel wird man jedoch $\alpha = 0.5$ wählen, so dass beide Geschichten gleich gewichtet sind.

Das Verhalten des exponentiellen Durchschnitt wird recht gut sichtbar, wenn die rekursive Formel expandiert wird. Dann ergibt sich:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1}\tau_0$$

Da α und $(1 - \alpha)$ kleiner oder gleich 1 sind, hat also jeder nachfolgende Term weniger Gewicht als sein Vorgänger.

Frage 5.15: Wie kann man SJF preemptiv machen?

SJF kann preemptiv oder nicht-preemptiv sein. Kommt ein neuer Prozess in die Ready Queue, dessen Burst-Zeit kürzer ist als die verbleibende des Prozesses der sich momentan auf der CPU befindet, so wird dieser preemptiv von der CPU genommen und der andere Prozess gestartet. In diesem Falle wäre dies also preemptives SJF (auch **Shortest-Remaining-Time-First** genannt). Im anderen Fall dürfte der Prozess auf der CPU seinen CPU-Burst beenden. Seien z.B. die folgenden Prozesse mit ihren Ankunftszeiten und Burst-Zeiten gegeben:

Prozess	Ankunftszeit	Burst-Zeit
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Dann würden diese wie in Abbildung 12 geschedult werden.

Frage 5.16: Inwiefern ist SJF ein Spezialfall des Priority Scheduling?

Abbildung 13: Priority Scheduling

Der SJF Algorithmus ist ein Prioritätsalgorithmus, bei dem die Priorität eines Prozesses das Inverse seiner (vorhergesagten) CPU-Burst-Zeit ist. Je länger der Burst, desto niedriger die Priorität und umgekehrt.

Frage 5.17: Was ist „Priority Scheduling"?

Wie schon gezeigt wurde, ist das SJF Scheduling ein Spezialfall der allgemeineren Klasse des Priority Scheduling. Beim Priority Scheduling wird mit jedem Prozess eine Priorität assoziiert und die CPU an den Prozess mit der höchsten Priorität gegeben. Prozesse mit gleicher Priorität werden mit FCFS Scheduling behandelt. Seien z.B. die folgenden Prozesse mit ihren Burst-Zeiten und Prioritäten gegeben, dann würden diese wie in Abbildung 13 geschedult werden.

Prozess	Burst-Zeit	Priorität
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Die Priorität eines Prozesses wird im allgemeinen durch einer Zahl (z.B. zwischen 0 und 7 oder zwischen 0 und 1023) angegeben, wobei eine hohe Priorität entweder durch eine hohe Zahl oder eine niedrige Zahl repräsentiert wird.

Frage 5.18: Wie können Prioritäten vergeben werden?

Prioritäten können intern oder extern vergeben werden. In ersterem Fall wird die Priorität z.B. durch messbare Quantitäten oder Qualitäten (z.B. Speicherbedarf, Zeitbeschränkung, etc.) errechnet und dem Prozess zugeteilt. In zweiterem Fall kann die Priorität z.B. durch den Benutzer oder Operator vorgegeben werden.

Frage 5.19: Wie funktioniert preemptives Priority Scheduling?

Beim preemptiven Priority Scheduling wird ein Prozess vorzeitig von der CPU genommen, sobald ein Prozess mit einer höheren Priorität in der Ready Queue ist. Bei nicht-preemptiven Scheduling darf der Prozess mit der niedrigeren Priorität erst seinen CPU-Burst beenden.

Frage 5.20: Welches Problem gibt es beim „Priority Scheduling"? Wie kann es gelöst werden?

Eines der Hauptprobleme des Priority Scheduling, wie es bis hier vorgestellt wurde, ist, dass ein Prozess „verhungern" kann (**Starvation** oder **Indefinite Blocking**). Dies bedeutet, dass ein Prozess mit einer niedrigen Priorität bereit wäre, auf der CPU ausgeführt zu werden, dabei jedoch immer wieder von Prozessen mit einer höheren Priorität verdrängt wird.

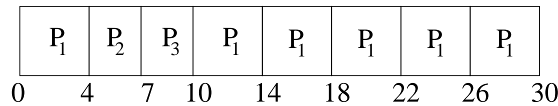


Abbildung 14: Round Robin-Scheduling

Eine Lösung dieses Problems ist es, die Prozesse altern zu lassen (**Aging**). Dies bedeutet, dass die Priorität eines Prozesses nach und nach steigt, je länger er wartet. Spätestens wenn der Prozess irgendwann die höchste Priorität hat, wird er laufen, da in diesem Fall bei mehreren Prozessen mit der gleichen Priorität mit der FCFS-Politik vorgegangen wird.

Frage 5.21: Wie funktioniert Round-Robin Scheduling?

Das Round-Robin Scheduling wurde speziell für Time-Sharing Systeme entwickelt. Es ist dem FCFS Scheduling recht ähnlich, jedoch ist es preemptiv. Dabei wird ein kleiner Zeitschlitz definiert, welcher üblicherweise zwischen 10 und 100 Millisekunden liegt. Die Ready Queue wird wie eine kreisförmige Queue behandelt, die der CPU Scheduler durchläuft. Dabei darf jeder Prozess so lange auf der CPU ausgeführt werden, bis er entweder seinen CPU-Burst beendet hat, oder bis der Zeitschlitz abgelaufen ist.

Die Ready Queue wird als FIFO-Queue implementiert, wobei jeweils der Prozess am Kopf der Queue auf die CPU darf und neue Prozesse ans Ende der Queue angehängt werden. Bei einem Context Switch wird der Prozess, der sich auf der CPU befindet, wieder ans Ende der Queue gepackt, sofern er nicht vorher beendet wurde. Seien z.B. die folgenden Prozesse mit ihrer Burst-Zeit gegeben:

Prozess	Burst-Zeit
P_1	24
P_2	3
P_3	3

Dann würden diese bei einem Zeitschlitz von 4 wie in Abbildung 14 geschedult werden.

Frage 5.22: Wie lange muss ein Prozess beim Round-Robin Scheduling maximal warten, bis er wieder ausgeführt wird?

Wenn ein Zeitschlitz q lang ist und es n verschiedene Prozesse gibt, muss ein Prozess maximal $(n-1) \cdot q$ Zeitschlitze warten, bis er wieder maximal einen Zeitschlitz lang ausgeführt wird.

Frage 5.23: Von welchem Parameter hängt die Performanz des Round-Robin Scheduling besonders ab?

Die Performanz des Round-Robin Scheduling hängt besonders stark von der Größe des Zeitschlitzes ab. Wird der Zeitschlitz zu groß gewählt, „verkommt“ der Round-Robin-Ansatz zu einem FCFS-Ansatz. Ist der Zeitschlitz extrem klein (z.B. eine Mikrosekunde) so spricht man von **Processor Sharing**. Dann hat bei n Prozessen jeder Prozess den Eindruck, er liefe alleine auf einer CPU, die nur $\frac{1}{n}$ der Geschwindigkeit der realen CPU aufweist. Dabei muss man jedoch bedenken, dass dieser Ansatz oft nur für Hardwarelösungen effizient ist.

In Software muss nämlich noch die Zeit bedacht werden, die für einen Context Switch benötigt wird.

Eine Faustregel für das Einstellen des Zeitschlitzes ist, dass 80% aller CPU-Bursts kürzer als ein Zeitschlitz sein sollten.

Frage 5.24: Wie funktioniert „Multilevel Queue Scheduling"?

Multilevel Queue Scheduling wurde für Situationen entworfen, in denen Prozesse in verschiedene Gruppen klassifiziert werden können. Eine übliche Aufteilung ist beispielsweise die in **Foreground**- und **Background**-Prozesse. Vordergrundprozesse sind zum Beispiel interaktive Prozesse, während Batchprozesse eher Hintergrundprozesse sind.

Beim Multilevel Queue Scheduling wird die Queue in verschiedene separate Queues aufgeteilt. Jede Queue kann dabei ihren eigenen Scheduling-Algorithmus haben. So kann es z.B. sinnvoll sein, für Vordergrund- und Hintergrundprozesse verschiedene Scheduling-Methoden zu verwenden. Etwa Round-Robin für die Vordergrundprozesse und FCFS für die Hintergrundprozesse. Die verschiedenen Queues besitzen auch verschiedene Prioritäten, wobei ein Prozess in einer Queue mit niedriger Priorität nur dann laufen kann, wenn alle anderen Queues mit höherer Priorität leer sind.

Eine weitere Möglichkeit besteht darin, zum Beispiel den Vorder- und Hintergrund-Queues verschiedene Zeitschlitze zuzuordnen.

Frage 5.25: Gebe eine Aufteilung in verschiedene Queues für das „Multilevel Queue Scheduling" an!

Eine mögliche Aufteilung ist:

1. Systemprozesse
2. Interaktive Prozesse
3. Interactive editing processes (?)
4. Batchprozesse
5. Studentenprozesse

Frage 5.26: Wie funktioniert „Multilevel Feedback-Queue Scheduling"?

Beim Multilevel Queue Scheduling bewegen sich die Prozesse im allgemeinen nicht zwischen den verschiedenen Queues hin und her. Wenn es z.B. getrennte Queues für Vorder- und Hintergrundprozesse gibt, bewegen sich die Prozesse nicht von einer Queue in die andere, da sie ihre Eigenschaft ein Vorder- oder Hintergrundprozess zu sein ja nicht ändern. Diese Art des Scheduling hat zwar geringen Overhead, ist jedoch unflexibel.

Beim **Multilevel Feedback-Queue Scheduling** wird es den Prozessen nun ermöglicht, zwischen den Queues zu wechseln. Im allgemeinen ist ein Multilevel Feedback-Queue Scheduler durch die folgenden Eigenschaften charakterisiert:

- Die Anzahl der Queues.
- Der Scheduling Algorithmus für jede Queue.

- Die Methode, welche benutzt wird zu entscheiden, wann ein Prozess in eine Queue mit einer höherern Priorität wandert.
- Die Methode, welche benutzt wird zu entscheiden, wann ein Prozess in eine Queue mit einer niedrigeren Priorität wandert.
- The method used to determine which queue a process will enter when that process needs service. (?)

Multilevel Feedback-Queue Scheduling besitzt die allgemeinste Definition aller Scheduling Algorithmen, ist daher jedoch auch am komplexesten von allen.

5.4 Scheduling mit mehreren Prozessoren

Frage 5.27: Was muss man beim Scheduling mit mehreren Prozessoren beachten?
In einem Mehrprozessor-System ist das Scheduling noch komplexer als in einem Einprozessor-System. Wenn z.B. ein I/O-Gerät an den Bus eines Prozessors angeschlossen ist, auf welchen nur dieser eine Prozessor zugreifen kann, dann müssen Prozesse, welche dieses Gerät benutzen möchten auch auf diesem Prozessor laufen.

Ein weiteres Problem besteht darin, dass beim symmetrischen Multiprocessing die Prozessoren auf eine gemeinsame Ready Queue zugreifen müssen, wenn man verhindern will, dass ein Prozessor eventuell garnichts zu tun hat, während ein anderer stark ausgelastet ist. In Kapitel 6 (Prozess-Synchronisation) wird auf die Probleme, die beim Zugriff auf gemeinsame Daten entstehen können, näher eingegangen.

Beim asynchronen Multiprocessing ist dieses Problem nicht ganz so gravierend, da hier nur der Master-Prozessor auf die Queue und die Geräte zugreift. Die Slave-Prozessoren führen nur User-Code aus.

Frage 5.28: Was muss beim Real-Time Scheduling beachtet werden?

5.5 Evaluation von Scheduling Algorithmen

Frage 5.29: Welche verschiedenen Methoden gibt es, um Scheduling Algorithmen zu evaluieren?

Deterministic Modeling: Die **deterministische Modellierung** ist ein analytisches Verfahren. Bei dieser Methode wird ein vorbestimmter Workload (Ankunft der Prozesse, ihre Burst-Zeiten, etc.) bestimmt und anhand dessen die Performanz der einzelnen Algorithmen verglichen.

Deterministisches Modellieren gibt exakte Werte, die jedoch immer nur für den betrachteten Fall zutreffen. Bei der Betrachtung mehrerer Beispiele können jedoch oft Trends abgelesen werden.

Queueing Models: Die Prozesse, die täglich auf einem System laufen, sind nicht immer die gleichen. Daher ist eine statische Betrachtung, wie beim „Deterministic Modeling“ nicht immer sinnvoll. Man kann jedoch die Verteilungen der CPU-Bursts und der I/O-Bursts ermitteln. Diese kann man in Formeln packen, die die Wahrscheinlichkeit für einen bestimmten Burst angeben. Diese Formeln sind meistens exponentiell. Genauso kann man die Verteilung ermitteln, wie Prozesse im System ankommen (**arrival-time distribution**) und aus diesen beiden Verteilungen kann man dann den durchschnittlichen Durchsatz, Wartezeit, etc. bestimmen. Diese Methode funktioniert mit den meisten Scheduling Algorithmen.

Simulationen: Bei dieser Methode wird das verwendete Computersystem komplett durch Software simuliert. Die Daten, die diese Simulation zu steuern, können dabei auf unterschiedliche Arten und Weisen erzeugt werden. Eine recht geläufige Methode ist die Verwendung eines Zufallszahlen-Generators, der Prozesse, mit ihren dazugehörigen Burst-Zeiten, Ankunftszeiten und anderen Charakteristiken generiert. Die Verteilungen der Wahrscheinlichkeiten können dabei entweder mathematisch vorgegeben sein (uniform, exponentiell, poissonverteilt) oder empirisch. Desweiteren können **Trace Tapes** verwendet werden, welche die Ereignisse eines realen Systems aufzeichnen, um damit später die Simulation zu speisen.

Implementierung: Selbst Simulation können zu einem gewissen Grad noch zu ungenau sein. Daher ist die einzig wirklich genaue Möglichkeit die Leistung eines Algorithmus zu bewerten, diesen zu implementieren und im realen System zu benutzen. Allerdings ist dies mit hohen Kosten verbunden. Zum einen kostet das Programmieren und die Modifikation des Betriebssystems Geld und Zeit. Desweiteren wird das System vielleicht auch für den Benutzer schlechter benutzbar, da es sich ständig ändert.

Frage 5.30: Wie lautet „Little's Formel“?

Little's Formel lautet:

$$n = \lambda \times W$$

Dabei ist n die durchschnittliche Länge der Queue (ohne den Prozess auf der CPU), λ ist die durchschnittliche Ankunftszeit neuer Prozesse und W die durchschnittliche Wartezeit. Littles Formel gilt dabei nur, wenn sich das System in einem stabilen Zustand befindet, wenn also die Anzahl der Prozesse, die die Queue verlassen, gleich der Anzahl der Prozesse ist, die neu hinzu kommen.

6 Prozess-Synchronisation

6.1 Einleitung

Frage 6.1: Warum wird Prozess-Synchronisation benötigt?

Wie wir in Kapitel 4 gesehen haben, können sich Prozesse bzw. mehrere Threads in einem Prozess Daten, wie z.B. Variablen, teilen. Es kann nun passieren, dass die einzelnen Threads Methoden bzw. Operationen ausführen, die für sich gesehen zwar funktionieren, jedoch nicht mehr, wenn die Threads nebenläufig ausgeführt werden.

Angenommen zwei Threads teilen sich die Variable `count` (aus dem Beispiel zum Producer-Consumer-Problem), welche den momentanen Wert 5 hat. Führt nun ein Thread ein `++count` aus und der andere ein `--count` so kann der Wert der Variable am Ende 4,5 oder 6 sein! Dies liegt daran, dass die Operation `++count` in Maschinensprache so ausgeführt werden kann:

$$\begin{aligned} ®ister_1 = \texttt{count}; \\ ®ister_1 = register_1 + 1; \\ &\texttt{count} = register_1; \end{aligned}$$

Dabei ist $register_1$ ein lokales CPU-Register. Und entsprechend kann `--count` wie folgt aussehen:

$$\begin{aligned} ®ister_2 = \texttt{count}; \\ ®ister_2 = register_2 \text{ - } 1; \\ &\texttt{count} = register_2; \end{aligned}$$

Wenn nun zwei Threads, z.B. ein Produzent und ein Konsument, die Methode `++count` und `--count` nebenläufig ausführen, bedeutet dies, dass die Maschinensprachinstruktionen irgendwie verschachtelt ausgeführt werden (aber immer noch in derselben Reihenfolge). Eine solche Verschachtelung könnte die folgende sein, wobei davon ausgegangen wird, dass der Produzent die Operation `++count` ausführt und der Konsument die Operation `--count`: Der Produzent führt die ersten beiden Maschineninstruktionen durch und danach der Konsument seine beiden ersten. Nun enthält $register_1$ den Wert 6 und $register_2$ den Wert 4. Anschließend führt der Produzent die Zuweisung `count` = $register_1$ durch, gefolgt vom Konsumenten, der `count` = $register_2$ setzt. Da $register_2$ jedoch den Wert 4 enthält, ist der Wert von `count` am Ende 4, obwohl er 5 sein müsste. Entsprechend kann der Wert 6 erreicht werden, wenn die Reihenfolge der beiden letzten Instruktionen vertauscht wird.

Der Grund hierfür ist, dass die drei Zeilen Maschinencode **atomar**, d.h. in einem Rutsch, ausgeführt werden müssten, da sonst der Wert einer Variablen manipuliert werden kann, ohne dass dieser neue Wert in den Speicher geschrieben wird. Greift nun aber ein anderer Thread auf diese Variable zu, so erhält er immer noch den alten Wert aus dem Speicher.

Frage 6.2: Was ist eine „Race Condition“?

Eine **Race Condition** ist eine Situation, in der z.B. mehrere Threads nebenläufig dieselben Daten manipulieren und das Ergebnis dieser Manipulation von der Reihenfolge der Zugriffe auf die Daten abhängt. Um eine Race Condition zu vermeiden, müssen die Threads in irgendeiner Art synchronisiert werden.

6.2 Das Critical-Section Problem

Frage 6.3: Was ist das „Critical-Section Problem“?

Gegeben seien n Threads $\{T_0, T_1, \ldots, T_{n-1}\}$. Damit beim Zugriff auf eine gemeinsame, geteilte Resource kein Chaos entsteht (z.B. Race Condition), wird Code, der auf diese Resource zugreift, als kritisch (**critical**) ausgezeichnet. Eine solche Codesektion wird **Critical Section** genannt. Dies kann zum Beispiel Code sein, der den Wert einer Variablen ändert, der eine gemeinsam genutzte Datenstruktur aktualisiert, der eine Datei schreibt, etc. Eine wichtige Eigenschaft des Systems ist es nun, dass wenn ein Thread eine Critical Section ausführt, kein anderer Thread sich ebenfalls in der Ausführung einer Critical Section befindet (*„[...] the execution of critical sections by the threads is mutually exclusive in time.“*). Um in seine Critical Section ausführen zu können, muss der Thread also um Erlaubnis fragen. Dies wird in einer **Enter**-Sektion gemacht. Nach der Critical Section kann eventuell noch eine **Exit**-Sektion kommen. Der Code nach der Critical Section ist die **Remainder**-Sektion.

Das **Critical-Section Problem** besteht nun darin, ein Protokoll zu finden, so dass die Threads kooperieren können. Dabei müssen drei Kriterien erfüllt sein:

Mutual Exclusion: Wenn Thread T_i sich in seiner Critical Section befindet, kann sich keiner der anderen Threads in seiner Critical Section befinden.

Progress: Wenn sich kein Thread in seiner Critical Section befindet und Threads existieren, die in ihre Critical Section möchten, nehmen nur die Threads an der Entscheidung teil, die nicht momentan ihre nicht-kritische Sektion ausführen. Es nehmen also quasi nur Threads teil, die schon „am Eingang zur Critical Section stehen“. Desweiteren kann diese Auswahl zeitlich nicht beliebig verschoben werden.

Bounded Waiting: Die Anzahl der Prozesse, die vor einem anderen Prozess ihre Critical Section ausführen, nachdem dieser eine Anfrage gemacht hat, seine Critical Section ausführen zu dürfen, ist beschränkt. Diese Forderung soll das „Verhungern“ (**Starvation**) eines Threads verhindern.

Frage 6.4: Gebe einen Algorithmus an, der das „Critical Section Problem“ für zwei Prozesse löst und dabei alle drei der geforderten Bedingungen erfüllt!

Wir werden das Problem in drei Schritten lösen. Dazu werden drei Algorithmen vorgestellt, wobei der letzte der drei eine Kombination der ersten beiden Algorithmen ist und als einziger alle drei Bedingungen erfüllt. Die Prozesse heißen P_0 und P_1. Es sei angemerkt, dass für den Prozess P_i jeweils P_{1-i} der andere Prozeß ist.

Algorithmus 1: Die Hauptidee dieses Algorithmus ist es, sich in einer Variablen zu merken, welcher Thread als nächstes seine Critical Section ausführen darf (z.B. über eine Variable `turn`). Der Pseudo-Code sähe wie folgt aus:

```
while( turn != i ) no-op;    // nicht dran -> nichts machen
```

```
criticalSection();
turn = 1-i;                        // der andere ist dran
```

Dabei wird jedoch leider die zweite Forderung (Progress) nicht erfüllt. Denn wenn einer der beiden Threads seine Critical Section ausführen möchte, jedoch nicht an der Reihe ist, so kann er seine Critical Section auch dann nicht ausführen, wenn der andere Thread sich gerade in der Ausführung seiner nicht-kritischen Sektion befindet.

Algorithmus 2: Das Problem bei Algorithmus 1 ist, dass er sich nicht merkt, in welchem Zustand sich welcher Thread gerade befindet. Daher ersetzen wir die Variable `turn` durch ein Array `boolean[] flag = new boolean[2]`. Zu Beginn wird das Array auf `false` initialisiert. Ein Thread T_i setzt `flag[i]` auf `true`, wenn er bereit ist, seine Critical Section auszuführen. Anschliessend schaut er nach, wie das Flag des anderen Thread gesetzt ist. Ist dies `false`, kann der Thread seine Critical Section ausführen und anschließend sein Flag wieder auf `false` setzen, damit der andere Thread seine Critical Section ausführen kann, falls er dazu bereit ist. Der Pseudo-Code dazu:

```
flag[i] = true;                        // wir wollen in die cs
while ( flag[1-i] == true ) no-op; // der andere ist drin
criticalSection();
flag[i] = false;                       // wir sind raus
```

Bei diesem Algorithmus ist die Mutual Exclusion-Bedingung erfüllt, nicht jedoch die Progress-Bedingung. Es kann nämlich passieren, dass Prozess P_i in der ersten Zeile `flag[i]` auf `true` setzt, dann der andere Prozess dran kommt, sein `flag[1-i]` ebenfalls auf `true` setzt und dann beide für immer in der `while`-Schleife verweilen.

Algorithmus 3: Dieser Algorithmus erfüllt beide Bedingungen und ist eine Kombination der beiden ersten Algorithmen, d.h. er benutzt eine Variable `turn` und ein Array `flag`. Will ein Thread in seine Critical Section, setzt er sein Flag auf `true` und setzt `turn` auf den anderen Thread. In einer `while`-Schleife wird nun überprüft, ob das Flag des anderen Thread auf `true` gesetzt ist und ob der `turn` immer noch den Wert des anderen Threads besitzt. Trifft dies *beides* zu (`&&`), führt der Thread ein `no-op` aus, ansonsten kann er seine Critical Section ausführen. Nach der Ausführung der Critical Section setzt der Thread sein Flag wieder auf `false`. Der Code:

```
flag[i] = true;
turn = 1-i;
while ( (flag[1-i] == true) && (turn = 1-i) ) no-op;
criticalSection();
flag[i] = false;
```

Die Bedingung der `while`-Schleife lässt sich leicht merken, wenn man sich fragt, wann ein Prozess *nicht* in seine Critical Section darf: Wenn der andere Prozess dran ist (`turn == 1-i`) und dieser auch in seine Critical Section möchte (`flag[1-i] == true`). In allen anderen Fällen darf der Prozess in seine Critical Section.

Frage 6.5: Gebe einen Algorithmus an, der das Critical Section-Problem für mehr als zwei Prozesse löst!

Gesucht ist ein Algorithmus, welcher für n Prozesse das Critical Section-Problem löst. Dies leistet der sogenannte **Bakery**-Algorithmus. Er funktioniert wie die Koordination auf Ämtern. Jedem Prozess, der in seine Critical Section will, wird eine Nummer zugeteilt und die Prozesse dann nach aufsteigenden Nummern „bedient“. Es kann dabei jedoch nicht gewährleistet werden, dass zwei Prozesse nicht dieselbe Nummer zugeteilt bekommen. In diesem Fall wird der Prozess mit der kleineren Prozess-ID zuerst bedient, so dass auch in diesem Fall eine vollständige Ordnung vorliegt, da keine zwei Prozesse die gleiche Prozess-ID haben können. Es werden zwei Datenstrukturen benötigt:

- Ein Boolean-Array, welches speichert, ob ein Prozess gerade eine Nummer zugeteilt bekommt: `boolean choosing[n]`
- Ein Integer-Array, welches die gezogenen Nummern der Prozesse speichert: `int number[n]`

Zu Beginn werden die beiden Datenstrukturen auf `false` bzw. auf Null initialisiert. Desweiteren wird eine Funktion $\max(a_0, \ldots, a_{n-1})$ definiert, welche eine Nummer k liefert die größer oder gleich jedem a_i für $i = 0, \ldots, n-1$ ist, also:

$$\max(a_0, \ldots, a_{n-1}) = k \text{ mit } k \geq a_i \text{ für } i = 0, \ldots, n-1$$

Algorithmus 1 zeigt den Ablauf des Algorithmus.

Algorithmus 1 Der Bakery-Algorithmus für Prozess P_i

```
choosing[i] = true;
number[i] = max(number[0], number[1], ..., number[n-1]) + 1;
choosing[i] = false;
for j=0 to n-1 do
  while choosing[j] do
    no-op;                     {Ein Prozess könnte noch vor uns landen}
  end while
  while choosing[j] ≠ 0 and (number[j],j) < (number[i],i) do
    no-op;                     {Es ist noch ein Prozess vor uns dran}
  end while
end for
criticalSection();
number[i]=0;
remainder();
```

Im ersten Schritt zieht P_i also eine Nummer, welche größer ist als alle anderen bisher vorhandenen Nummern. Dann wird für jeden anderen Prozess P_j getestet, ob er eventuell noch vor Prozess P_i dran ist. Die erste while-Schleife überprüft, ob ein Prozess seine Nummer noch am ziehen ist. Es kann ja passieren, dass bei der Zuweisung des Ergebnisses von max auf number[j] ein Interrupt auftritt und der Prozess P_j von der CPU runtergenommen wird. Dann ist P_i dran, zieht seine Nummer und läuft bis in die for-Schleife. Kommt P_j nun wieder auf die CPU bekommt er seine Nummer, welche kleiner als unsere ist. Dieser Fall muss also beachtet werden, da ansonsten bei der zweiten while-Schleife ein Fehler passieren würde. Aus diesem Grund wird gewartet, wenn ein Prozess eine Nummer zieht. Die zweite while-Schleife überprüft danach einfach, ob ein Prozess P_j überhaupt gerade in seine Critical Section will (number[j]$\neq$0) und wenn dies der Fall ist, ob seine Nummer eventuell kleiner ist, d.h. ob er vor P_i dran ist.

6.3 Synchronisationshardware

Frage 6.6: Wie kann Synchronisation mit Hilfe von Hardware unterstützt werden?

Auch im Bereich der Synchronisation können einige Änderungen an der Hardware, die Lösung des Critical Section-Problem erheblich vereinfachen. In einem Einprozessor-System können während der Ausführung einer Critical Section die Interrupte einfach ausgeschaltet werden und nach der Beendigung der Critical Section wieder an. In einem Multiprozessor-System ist dies jedoch etwas umständlicher, da bei diesem Vorgang *jeder* Prozessor benachrichtigt werden muss. Durch eine der beiden folgenden Operationen auf Hardware-Ebene kann dies jedoch vereinfacht werden:

testAndSet: Eine Möglichkeit, Threads zu synchronisieren, besteht darin, in einer booleschen Variable `lock`, auf die alle Threads zugreifen können, zu speichern, ob sich ein Thread momentan in seiner Critical Section befindet. Algorithmus 2 zeigt eine naive Möglichkeit, die Synchronisation mittels einer geteilten Variable zu realisieren. Das Problem hierbei ist jedoch, dass nicht gewährleistet ist, dass das Testen und Setzen der Variable `lock` ohne Unterbrechung ausgeführt wird. Über Hardware kann nun eine Operation angeboten werden, welche eine Variable testet und setzt. Diese Operation wird atomar ausgeführt. Die Verwendung einer solchen Operation wird in Algorithmus 3 dargestellt.

swap: Angenommen die Hardware stellt eine Operation `swap(a,b)` zur Verfügung, die atomar den Inhalt der beiden Variablen `a` und `b` vertauscht. Dann kann diese Operation wie in Algorithmus 4 genutzt werden, um Threads bzw. Prozesse zu synchronisieren. Bevor ein Thread in seine Critical Section geht, setzt er seinen eigenen `key`-Wert auf `true` und holt sich über die `swap`-Operation eine Kopie der Variable `lock`. Besaß die `lock`-Variable zum Zeitpunkt des Austauschs den Wert `false`, was bedeutet, dass sich kein anderer Thread in seiner Critical Section befand,

Algorithmus 2 Critical Section ohne test-and-set

```
if lock is not set then
   set the lock;
   enter critical section;
   ...
   leave critical section;
   unset the lock;
else
   ...
end if;
```

Algorithmus 3 Critical Section mit test-and-set

```
if test the lock and set the lock then
   enter critical section;
   ...
   leave critical section;
   unset the lock;
else
   ...
end if;
```

so ist sie nach dem Austausch `true`. Selbst wenn sofort im Anschluss ein anderer Prozess (Thread) eine `swap`-Operation durchführt, erhält er den Wert `true` zurück, da `swap` atomar durchgeführt wird und bleibt in seiner busy-wait `while`-Schleife. Der Prozess, der als erstes beim Austausch erfolgreich war, verlässt seine `while`-Schleife und betritt seine Critical Section. Anschliessend setzt er die `lock`-Variable auf `false`.

Algorithmus 4 Critical Section mit swap

```
shared boolean lock = false;
boolean key = true;
while true do
   key.set(true);
   while key.get() == true do
      swap(lock, key);
   end while;
   enter critical section;
   lock.set(false);
   leave critical section;
end while;
```

6.4 Semaphoren

Frage 6.7: Was ist eine Semaphore?

Eine Semaphore S ist eine Integer-Variable, die nur über zwei Operationen `P(S)` und `V(S)` manipuliert werden kann. Diese sehen wie folgt aus:

```
P(S) {
   while( S <= 0 )
     ;  // no-op

   S--;
}

V(S) {
   S++;
}
```

Dabei müssen die Operationen in `P` und `V`, die die Semaphore manipulieren und abfragen (also alles außer das `no-op`), ohne Unterbrechung durchgeführt werden. Dies kann z.B. durch Ausschalten der Interrupte erreicht werden. Prinzipiell kann die Unterdrückung von Interrupten problematisch werden, bei Semaphoren geht man jedoch davon aus, dass der betroffene Code hinreichend kurz ist, so dass die Interrupte nicht lange ausgeschaltet sind.

In einem Multiprozessorsystem lässt sich das oben beschriebene Verfahren der Unterdrückung von Interrupten nicht effizient realisieren. Es ist eine Nachricht an jeden einzelnen Prozessor zu versenden, welche diese über die Unterdrückung von Interrupten benachrichtigt. Hier könnten Hardware-Lösungen Anwendung finden.

Frage 6.8: Wie kann eine Semaphore im Zusammenhang mit dem Critical Section-Problem benutzt werden?

Allgemein wird eine Semaphore wie folgt benutzt, um das Critical-Section Problem zu lösen:

```
Semaphore S;

P(S);
criticalSection();
V(S);
```

Es kann zwischen binären (**binary**) und zählenden (**counting**) Semaphoren unterschieden werden. Zählende Semaphoren werden benutzt, um Zugriff auf eine Resource zu ermöglichen, welche nur in begrenzter Anzahl zur Verfügung steht. Die Semaphore wird dabei auf die Anzahl der zur Verfügung stehenden Resourcen initialisiert. Jeder Thread, der die Resource benutzen möchte, erniedrigt die Semaphore um Eins. Ist der Wert gleich Null, sind alle Resourcen benutzt. Diese Methode wird etwas weiter im Text benutzt werden, um das Producer-Consumer-Problem zu lösen.

Frage 6.9: Wie kann eine Semaphore benutzt werden um zwei Prozesse zu synchronisieren?

Angenommen es gibt eine Sektion $S1$ in einem Prozess P_1, die auf jeden Fall vor einer Sektion $S2$ in einem Prozess P_2 ausgeführt werden soll. Dann kann dies

wie folgt bewerkstelligt werden. Benutzt wird eine Semaphore `synch`, welche auf 0 initialisiert ist. In den Code von P_1 wird nun das folgende Segment eingefügt:

```
S1;
V(synch);
```

In Prozess P_2 wird der folgende Code eingefügt:

```
P(synch);
S2;
```

Frage 6.10: Was versteht man unter „Busy-Waiting"? Warum ist dies in einem Einprozessor-System unerwünscht?

Unter **Busy-Waiting** versteht man das kontinuierliche Verbleiben in einer `while`-Schleife, solange eine Bedingung erfüllt ist, ohne dabei etwas Sinnvolles zu tun. Ein Beispiel hierfür ist die `while`-Schleife der `P(S)`-Operation. Während dies in einem Einprozessor-System CPU-Zyklen verbraucht, die eventuell für sinnvollere Arbeit benutzt werden könnten, kann Busy Waiting in einem Mehrprozessor-System jedoch ökonomischer sein. Der Grund hierfür liegt darin, dass die anderen Prozessoren weiterarbeiten können und gegenüber einem Context Switch Zeit gespart werden kann, wenn der Lock nicht allzu lange gesetzt ist (Stichwort: **Spinlock**).

Frage 6.11: Wie können Semaphoren modifiziert werden, so dass sie ohne „Busy Waiting" auskommen?

Der ursprüngliche Code für die Semaphoren muss wie folgt geändert werden:

```
P(S) {
   value--;
   if( value < 0 ) {
     add this process to list;
     block;
   }
}

V(S) {
   value++;
   if( value <= 0 ) {
     remove a process P from list;
     wakeup(P);
   }
}
```

Dabei sind `block` und `wakeup` System Calls. Ein Prozess, der `block` aufruft, belastet nicht die CPU, da er in eine zur Semaphore gehörende Queue gepackt wird und dort erst wieder rausgeholt wird, wenn ein anderer Prozess ihn mittels der Operation `V(S)` mit Hilfe des System Calls `wakeup` aufweckt.

Frage 6.12: Wie kann eine zählende Semaphore mit Hilfe von binären Semaphoren implementiert werden?

Um eine zählende Semaphore mit Hilfe von binären Semaphoren zu implementieren, werden eine Integer-Variable `count` und zwei binäre Semaphoren `countLock` und `semLock` benutzt:

count: Speichert jeweils den aktuellen Wert der zählenden Semaphore. Zu Beginn auf den Startwert der zählenden Semaphore initialisiert.

countLock: Wird benutzt, um den Zugriff auf `count` zu schützen. Zu Beginn auf 1 initialisiert.

semLock: Wird benutzt, um Prozesse zu blocken, die `count` auf einen Wert kleiner als Null setzen und damit warten müssen. Auf 0 initialisiert.

Diese drei Variablen werden wie in Algorithmus 5 und 6 benutzt. Bei beiden Operationen wird zu Beginn jeweils die Variable `count` geschützt und entsprechend der Operation aktualisiert. Bei der P()-Operation wird dann als nächstes geschaut, ob `count` kleiner als Null ist. Ist dies der Fall, wird den anderen Prozessen wieder die Chance gegeben, `count` zu manipulieren und der Prozess legt sich über semLock.P() quasi schlafen. Dies funktioniert, da semLock auf Null initialisiert ist, d.h. der aufrufende Prozess wird solange in seiner while-Schleife verweilen bzw. geblockt werden, bis ein anderer Prozess semLock.V() aufruft. Denn wenn ein Prozess semLock.V() aufruft und sich einer oder mehrere Prozesse in ihrem Busy Wait bzw. in der Block-Queue der Semaphore befinden, wird einer davon merken, dass die binäre Semaphore semLock auf Eins steht und seine while-Schleife verlassen bzw. einer von ihnen per `wakeup` System Call geweckt werden. Anschließend wird er semLock jedoch wieder auf Null setzen, so dass die anderen Prozesse weiter warten werden. In der V()-Operation wird geschaut, ob `count` ≤ 0 ist. Ist dies der Fall, wartet noch mindestens ein Prozess und einer von diesen wird per semLock.V() geweckt. Ansonsten wartet kein anderer Prozess und der Lock auf `count` wird einfach freigegeben. Er wird nur in diesem Fall freigegeben, da ein wartender Prozess in der P()-Operation zweimal countLock.V() ausführt und dies quasi für den V() aufrufenden Prozess übernimmt.

Algorithmus 5 Implementierung der P()-Operation

```
countLock.P();
count = count - 1;
if count < 0 then
   countLock.V();
   semLock.P();
end if
countLock.V();
```

6.5 Klassische Sychronisationsprobleme

Frage 6.13: Welche klassischen Synchronisationsprobleme kennst du?

- Bounded-Buffer Producer-Consumer Problem

Algorithmus 6 Implementierung der V()-Operation

```
countLock.P();
count = count + 1;
if count ≤ 0 then
  semLock.V();
else
  countLock.V();
end if
```

- Readers-Writers Problem
- Dining Philosopher Problem

Frage 6.14: Worin besteht das Bound-Buffer Problem?

Das Bounded-Buffer Problem ist das Producer-Consumer Problem mit einem beschränkten Puffer. Um das Problem mittels Semaphoren zu lösen, benutzen wir drei Semaphoren:

- `mutex`: eine binäre Semaphore, welche den Mutual Exclusion-Zugriff auf den Zähler gewährleistet.
- `empty`: eine Semaphore, die auf die Puffergröße initialisiert wird.
- `full`: eine Semaphore, die auf Null initialisiert wird.

Grob gesehen ist das Vorgehen wie folgt: In der `enter`-Methode wird die `P()`-Methode der `empty`-Semaphore ausgeführt. Ist der Puffer bereits voll, so muss der Produzent warten. Andernfalls wird der Mutual Exclusion-Zugriff auf die `counter`-Variable beantragt und im positiven Fall alle nötigen Operationen durchgeführt. Im Anschluss an die Ausführung werden die `V()`-Methoden der `mutex`-, sowie der `full`-Semaphore aufgerufen, so dass der Konsument seine Arbeit fortsetzen kann, bzw. geweckt wird. Der Ablauf der `remove`-Methode ist ähnlich, jedoch werden, neben den Aufrufen der `mutex`-Methoden, die `P()`-Methode der `full`-Semaphore, sowie die `V()`-Methode der `empty`-Semaphore ausgeführt.

Der Trick an der Sache ist, dass eine Semaphore hochgezählt werden muss, damit ein Ereignis nicht mehr gegeben ist (`full` für das Ereignis Puffer leer) und eine Semaphore runtergezählt werden muss, bis ein Ereigniss gegeben ist (`empty` für Puffer voll). Auf den ersten Blick scheint diese Namensgebung unlogisch zu sein, betrachtet man jedoch, von wem und an welcher Stelle die entsprechenden Semaphoren aufgerufen werden, so erscheint sie wieder logischer. Der Produzet prüft stets, ob noch Platz im Puffer vorhanden ist, der Konsument überprüft zuerst, ob sich überhaupt etwas im Puffer befindet. Eine gute Merkregel, um sich die Verwendung der Semaphoren zu merken, ist, sich einfach ein Fragezeichen hinter den Variablen `emtpy` und `full` vorzustellen.

Frage 6.15: Worin besteht das Readers-Writers Problem?

Angenommen beliebig viele Threads teilen sich eine Datenbank. Ein Teil dieser Threads sind **Reader**, der andere Teil sind **Writer**. Die Bedingungen für den

Algorithmus 7 Der Code des Produzenten

```
empty.P();
mutex.P()
produce();
counter = counter + 1;
mutex.V();
full.V();
```

Algorithmus 8 Der Code des Kosumenten

```
full.P();
mutex.P()
consume();
counter = counter - 1;
mutex.V();
empty.V();
```

Zugriff auf die Datenbank sehen wie folgt aus: Es dürfen beliebig viele Reader auf die Datenbank zugreifen, aber immer nur genau ein Writer darin schreiben. Dies jedoch auch nur unter der Voraussetzung, dass kein Reader auf die Datenbank Zugriff hat.

Gelöst wird das Problem mittels zwei Semaphoren und einer Zählvariable:

- `db`, eine binäre Semaphore, welche den Zugriff auf die Datenbank gemäß der obigen Bedingungen regelt.
- `mutex`, eine binäre Semaphore, welche den Mutual Exklusion-Zugriff auf `readCount` gewährleistet.
- `readCount`, ein Zähler, in welchem vermerkt ist, wieviele Reader auf die Datenbank zugreifen.

Reader und Writer konkurieren um die Datenbank. Greift beispielsweise zuerst ein Reader auf die Datenbank zu, so sperrt er sie mittels `db` für alle Writer. Weitere Reader dürfen auf die Datenbank zugreifen, müssen sich jedoch „anmelden“, indem sie `readCount` um Eins erhöhen. Verlässt ein Reader die Datenbank, so vermindert er `readCount` um Eins. War er der letzte Reader, so gibt er `db` frei. Greift nun ein Writer auf die Datenbank zu, so sperrt er sie mittels `db` für alle anderen Threads. Verlässt er die Datenbank, gibt er `db` wieder frei. Reader und Writer können abermals um den Zugriff konkurieren. Bei dieser Lösung kann es jedoch zum Verhungern von Readern bzw. Writern kommen.

Algorithmus 9 Der Code eines Writers

```
db.P();
write();
db.V();
```

Frage 6.16: Worin besteht das Dining Philosopher Problem?

Algorithmus 10 Der Code eines Readers

```
mutex.P();
readCount = readCount + 1;
if readCount == 1 then
    db.P();
end if
mutex.V();
read();
mutex.P();
readCount = readCount - 1;
if readCount == 0 then
    db.V()
end if
mutex.V();
```

Das Szenario beim Dining Philosophers Problem ist wie folgt: Fünf Philosophen sitzen an einem runden Tisch, jeder hat vor sich einen Teller stehen und rechts und links davon ein Essstäbchen. Dabei ist das linke Essstäbchen eines Philosophen das rechte Essstäbchen seines linken Nachbarn. Insgesamt gibt es also 5 Essstäbchen. Ein Philosoph kann dabei drei Zustände annehmen: denken, hungrig sein und essen. Wird ein Philosoph hungrig, so versucht er sein linkes und rechtes Essstäbchen aufzunehmen. Er kann jedoch nicht beide Essstäbchen gleichzeitig aufnehmen, sondern muss eines nach dem anderen nehmen. Eine weitere Bedingung ist, dass dieser Vorgang nicht unterbrochen werden kann (keine Preemption).

Wie kann es hier nun zu einem Deadlock kommen? Zum Beispiel wenn alle Philosophen auf einmal hungrig werden und diese der Reihe nach zuerst ihr rechtes Essstäbchen aufnehmen. In diesem Fall würde jeder Philosoph auf seinen linken Nachbarn warten.

6.6 Monitore

Frage 6.17: Was versteht man unter einem Monitor?

Semaphoren können ein gutes Synchronisationsmittel sein, sie müssen jedoch dazu auf jeden Fall in richtiger Art und Weise verwendet werden. Wenn zum Beispiel ein Programmierer erst `mutex.V()` aufruft und danach `mutex.P()` kann es passieren, dass sich zwei Prozesse gleichzeitig in ihrer Critical Section befinden. Oder es kann passieren, dass im Programm zweimal `mutex.P()` aufruft und sich so selbst deadlockt. Wird zweimal `mutex.V()` aufgerufen, können wiederum zwei Prozesse gleichzeitig ihre Critical Section ausführen.

Ein Monitor, vorausgesetzt er wurde korrekt implementiert, löst dieses Problem, indem er die Synchronisation nicht mehr dem Programmierer überlässt. Ein Monitor ist ein Konstrukt, welches von höheren Programmiersprachen zur Verfügung gestellt wird. Es ist die Aufgabe des Compilers, die Synchronisation zu realisieren.

Grob gesehen besteht ein Monitor aus Methoden, welche von außen sichtbar sind, die auf geteilten Variablen arbeiten. Diese geteilten Variablen können nicht direkt manipuliert werden, sondern nur durch die im Monitor definierten Methoden. Es kann sich immer nur ein Thread im Monitor befinden, d.h. eine Methode kann immer nur von einem Thread ausgeführt werden. Der Monitor sorgt also dafür, dass die Methodenaufrufe **mutual exclusive** sind, der Programmierer hat mit der Synchronisation also nichts direkt zu tun.

Es kann jedoch vorkommen, dass ein Programmierer die Synchronisation der Threads nicht ganz allein dem Monitor überlassen will. Dazu gibt es einen besonderen Variablentyp, die **Condition**-Variable. Diese bietet zwei Funktionen an: `signal` und `wait`.

Frage 6.18: Beschreibe die „Signal"- und „Wait"-Variablen für Monitore!

Angenmmen wir haben eine Condition `x` definiert, dann bewirkt der Aufruf der Operation `x.wait()`, dass sich der aufrufende Thread so lange schlafen legt, bis ein anderer Thread die Operation `x.signal()` aufruft. Wenn z.B. ein Thread auf das Ergebnis eines anderen Threads angewiesen ist, so legt er sich mit `x.wait()` schlafen und wird per `x.signal()` von dem anderen Thread wieder aufgeweckt, sobald das Ergebnis vorliegt.

Grundsätzlich können zwei Vorgehensweisen der Realisierung der `signal`-Operation unterschieden werden. Im folgenden ist Q ein Prozess, der `wait()` aufgerufen hat und P der Prozess, welcher `signal()` aufruft.

Signal-and-Wait: Nachdem P `signal()` aufruft, wartet er bis Q den Monitor verlässt oder wartet auf eine andere Bedingung.

Signal-and-Continue: Q wartet bis P den Monitor verlassen hat oder wartet auf eine andere Bedingung.

Beide Implementierungen sind möglich, wobei erstere Lösung aus beweistechnischen Gründen favorisiert wird. Außerdem kann es beim zweiten Ansatz passieren, dass die Bedingung auf die Q gewartet hat, nicht mehr vorliegt, da P eventuell weitere Änderungen vorgenommen haben kann.

Frage 6.19: Wie kann unter Nutzung von Monitoren das Dining Philosopher Problem gelöst werden?

Beschreibe Algorithmus 7.27 Seite 204.

7 Deadlocks

7.1 Einleitung

Frage 7.1: Was ist ein Deadlock?
In einem Computer-System gibt es unterschiedliche Resourcen, um welche die verschiedenen Prozesse konkurrieren. Es kann nun passieren, dass ein Prozess eine Resource anfordert, welche er nicht sofort bekommen kann. Er geht dann in einen Wartezustand über, aus dem er jedoch eventuell nie wieder zurückkehren kann, da die Resource von anderen wartenden Prozessen belegt ist. Am besten wird ein Deadlock vielleicht durch ein Gesetz aus dem frühen 20. Jahrhundert beschrieben: „Wenn sich zwei Züge an einer Kreuzung treffen, so müssen beide anhalten, und dürfen nicht eher weiterfahren, als bis dass der andere Zug weggefahren ist.“

Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, wenn jeder Prozess der Menge auf ein Ereignis wartet, dass nur durch einen anderen Prozess in dieser Menge ausgelöst werden kann.

Frage 7.2: Nenne einige Beispiele für Resourcen!
Man kann zwischen physikalischen und logischen Resourcen unterscheiden:

Physikalisch: Drucker, Monitor, CPU-Zyklen, Hauptspeicher, Plattenplatz, andere I/O-Geräte.

Logische Semaphoren, Monitore, Dateien

Frage 7.3: Wie werden Resourcen beim Betriebssystem beantragt?
Resourcen können mit Hilfe von System Calls beim Betriebssystem beantragt werden. Dies sind zum Beispiel: `request` und `release device` oder auch `open` und `close file` sowie `allocate` und `free memory`.

7.2 Characterisierung von Deadlocks

Frage 7.4: Welche Voraussetzungen müssen gegeben sein, damit ein Deadlock auftritt?
Es müssen vier verschiedene Voraussetzungen gegeben sein:

Mutual Exclusion: Es muss mindestens eine Resource im System geben, welche nur exklusiv von einem Prozess benutzt werden kann. Wenn ein anderer Prozess Zugriff auf diese Resource haben möchte, so muss er warten, bis diese frei wird.

Hold and Wait: Es muss einen Prozess geben, der mindestens eine Resource besetzt und auf eine andere Resource wartet, welche von einem anderen Prozess benutzt wird.

No Preemption: Resourcen können nur durch den Prozess, der sie hält, freigegeben werden, nachdem dieser seine Aufgabe erfüllt hat.

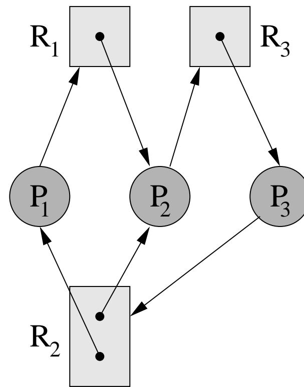


Abbildung 15: Beispiel für ein RAG (in einem Deadlock-Zustand)

Circular Wait: Es muss eine Menge von Prozessen $\{P_0, P_1, \ldots, P_n\}$ existieren, so dass P_0 auf eine Resource wartet, welche von P_1 gehalten wird, P_1 auf eine Resource, die von P_2 gehalten wird, ..., P_n auf eine Resource, die von P_0 gehalten wird.

Es ist wichtig, dass alle vier Bedingungen gleichzeitig gelten müssen, wobei sich diese teilweise gegenseitig implizieren. So impliziert die Bedingung **Circular Wait** die Bedingung **Hold-and-Wait**.

Frage 7.5: Was ist ein „Resource-Allocation Graph"?

Ein **Resource-Allocation Graph** ist ein Graph $G := \{V, E\}$, wobei sich V aus Prozess- und Resourcenknoten zusammensetzt.

- $P = \{P_1, P_2, \ldots, P_n\}$
- $R = \{R_1.R_2, \ldots, R_m\}$

Die Kantenmenge E unterteilt sich in:

Request Edges: Eine gerichtete Kante $P_i \to R_j$, die bedeutet, dass Prozess P_i Resource R_j angefordert hat.

Assignment Edges: Eine gerichtete Kante $R_j \to P_i$, die bedeutet, dass Prozess P_i Resource R_j belegt.

Zeichnerisch wird eine Resource durch ein Rechteck dargestellt, wobei mehrere Instanzen einer Resource durch Punkte in diesem Rechteck dargestellt werden. Prozesse werden durch Kreise dargestellt. Abbildung 15 zeigt wie so ein RAG aussehen kann. In diesem Fall befinden sich die Prozesse in einem Deadlock.

Frage 7.6: Wie kann mit Hilfe des „Resource-Allocation Graph" (RAG) festgestellt werden, ob ein Deadlock vorliegt oder nicht?

Ist der RAG kreisfrei, so ist das System frei von Deadlocks. Findet sich ein Kreis im RAG, so sind zwei Fälle zu unterscheiden:

- Es gibt von jeder Resource nur eine Instanz. In diesem Fall gibt es einen Deadlock. Dies wäre in Abbildung 16 der Fall, wenn Prozess P_2 nicht da wäre und es von R_1 nur eine Instanz gäbe.

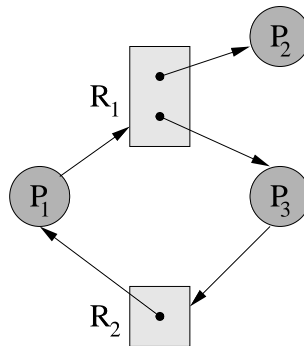


Abbildung 16: RAG mit einem Kreis ohne Deadlock

- Wenn es von jeder Resource mehr als eine Instanz gibt. so bedeutet ein Kreis noch nicht, dass ein Deadlock vorliegt. Hier ist ein Kreis eine notwendige aber nicht hinreichende Bedingung. Abbildung 16 zeigt ein Beispiel für einen solchen Fall. Wenn P_2 seine Instanz der Resource R_1 nicht mehr braucht, kann Prozess P_1 diese benutzen und wenn dieser mit seiner Aufgabe fertig ist, kann Prozess P_3 die angefragten Resourcen bekommen.

7.3 Methoden zum Umgang mit Deadlocks

Frage 7.7: Welche Methoden kennst du, mit mit Problem der Deadlocks umzugehen?
Es gibt drei verschiedene Ansätze:

Deadlock-Prevention (Deadlock-Verhinderung): Wie am Anfang dieses Kapitels gesehen, gibt es vier Bedingungen, die im Falle eines Deadlocks alle erfüllt sein müssen. Mit diesem Ansatz verfolgt man das Ziel, mindestens eine dieser Bedingungen auszuschalten.

Deadlock-Avoidance (Deadlock-Vermeidung): Bei diesem Ansatz besitzt das Betriebssystem von vornherein so viele Informationen über alle Prozesse, so dass es entscheiden kann, ob ein Prozess eine Resource zugeteilt bekommt oder nicht. Dabei können zwei Strategien verwendet werden:

- Es wird stets so entschieden, dass ein Deadlock erst garnicht auftreten kann.
- Deadlocks werden zugelassen, können jedoch erkannt und beseitigt werden.

Deadlocks ignorieren: Das Betriebssystem tut so, als ob Deadlocks nicht auftreten können. Viele Betriebssysteme, unter anderem UNIX, behandeln Deadlocks auf diese Art und Weise. Tritt ein Deadlock auf, so wird das System über kurz oder lang einfrieren, so dass es neu gestartet werden muss. Der Grund dafür, dass Deadlocks oft so behandelt werden, liegt darin, dass Deadlocks nicht oft auftreten (z.B. einmal im Jahr) und deshalb

die Kosten für einen Neustart geringer sind als die ständige algorithmische Überwachung des Systems.

7.4 Deadlock Prevention (Deadlock-Verhinderung)

Frage 7.8: Erläutere Methoden zur „Deadlock-Prevention"!
Bei der Vorgehensweise der Deadlock-Prevention wird versucht, eine der vier Bedingungen für einen Deadlock von vornherein auszuschließen. Es wird nun kurz auf alle vier Bedingungen eingegangen:

Mutual Exclusion: Die Idee hierbei wäre, alle Resourcen im System teilbar zu machen, d.h. dass mehrere Prozesse gleichzeitig auf diese zugreifen können. Da dies bei einigen Resourcen nicht gehen kann (Dateien, Drucker), ist dieser Ansatz im allgemeinen nicht nutzbar.

Hold-and-Wait: Die Idee besteht darin, entweder die „Hold"-Bedingung auszuschalten, so dass ein Prozess keine Resourcen belegt, während er neue Resourcen beantragt oder die „Wait"-Bedingung, so dass ein Prozess während seiner Ausführung nicht mehr auf neue Resourcen warten muss. Um dies zu realisieren, können die folgenden zwei Protokolle benutzen werden:

Hold: Jedesmal bevor ein Prozess neue Resourcen anfordert, muss er die alten alle abgeben.

Wait: Jeder Prozess muss zu Beginn seiner Ausführung *alle* Resourcen, die er später benötigt, beantragen. Dadurch kommt man also nie mehr in die Situation, dass der Prozess Resourcen hält und neue beantragt. Dies wird erreicht, indem alle System Calls, die Resourcen beantragen, zu Beginn des Programm kommen (macht wahrscheinlich der Compiler des Systems).

Beispiele zu beiden Protokollen gibt es im Buch. Die Nachteile sind:

- Die Resourcennutzung ist niedrig.
- Das Verhungern von Prozessen ist möglich.

No Preemption: Die Grundidee ist es, dass Resourcen, die einem Prozess schon zugeteilt wurden, unter bestimmten Umständen diesem Prozess noch entzogen werden können. Dabei gibt es zwei Möglichkeiten:

- Wenn ein Prozess, der schon einige Resourcen hält, weitere Resourcen benötigt und diese nicht sofort verfügbar sind, so werden alle Resourcen, die der nachfragende Prozess hält, diesem entzogen. Diese Resource werden in eine Liste der Resourcen eingetragen, auf die der Prozess wartet. Der Prozess wird erst dann wieder neu gestartet, wenn er sowohl seine alten als auch die neu angeforderten Resourcen bekommen kann.

- Dieser Fall ist ähnlich wie der erste. Wenn ein Prozess A Resourcen anfordert, die von einem anderen, wartenden Prozess B gehalten werden, werden diese dem Prozess B weggenommen. Jedoch nur sofern dieser auf irgendwelche anderen Resourcen wartet. Einem wartenden Prozess kann es also passieren, dass ihm Resourcen weggenommen werden. Er kann erst dann weiter ausgeführt werden, wenn er seine alten (weggenommenen) Resourcen wiederbekommen sowie die geforderten neuen Resourcen zugeteilt bekommt.

Dieses Protokoll wird oft bei Resourcen angewandt, die leicht gespeichert und wiederhergestellt werden können. Allerdings kann es hier zum Verhungern von Prozessen kommen, wenn diesen immer wieder Resourcen weggenommen werden.

Circular Wait: Eine Möglichkeit zyklisches Warten zu verhindern, ist es, eine totale Ordnung auf den Resourcen zu definieren. Das heißt, es wird eine Abbildung $F : R \rightarrow \mathbb{N}$ definiert, welche die Resourcen auf die natürlichen Zahlen abbildet.

Resourcen dürfen dann nur in steigender Folge angefordert werden. Zu Beginn kann ein Prozess eine beliebige Anzahl an Instanzen einer Resource R_i anfordern. Danach kann er nur noch Instanzen der Resourcen R_j anfordern, für die gilt: $F(R_j) > F(R_i)$. Möchte ein Prozess dabei mehrere Instanzen einer Resource anfordern, so muss er dies in *einer* Anfrage machen.

Alternativ kann verlangt werden, dass ein Prozess, wenn er eine Resource R_j anfordert, er alle Resourcen R_i freigibt, für die gilt: $F(R_i) \geq F(R_j)$.

Wenn beide Protokolle benutzt werden, kann es nicht zum zyklischen Warten kommen. Wir führen den Beweis als einen Beweis durch Widerspruch: Angenommen die Prozesse $\{P_0, P_1, \ldots, P_n\}$ würden in der Art und Weise aufeinander warten, dass P_i auf die Resource R_i wartet, welche von P_{i+1} gehalten wird. Da Prozess P_{i+1} die Resource R_i hält und R_{i+1} anfordert, folgt daraus, dass $F(R_i) < F(R_{i+1})$ gelten muss. Daraus folgt wiederum aufgrund der Transitivität:

$$F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$$

Da $F(R_0) < F(R_0)$ nicht gelten kann, kann also auch kein zyklisches Warten auftreten.

7.5 Deadlock Avoidance (Deadlock-Vermeidung)

Frage 7.9: Was versteht man unter „Deadlock Avoidance"?

Im Gegensatz zu Deadlock Prevention, wo man durch Ausschaltung einer der vier Bedingungen versucht, Deadlocks zu verhindern, geht man bei Deadlock Avoidance einen anderen Weg. Hier verfügt das Betriebssysteme über zusätzliche Informationen bezüglich des Resourcenbedarfs der Prozesse. Oft

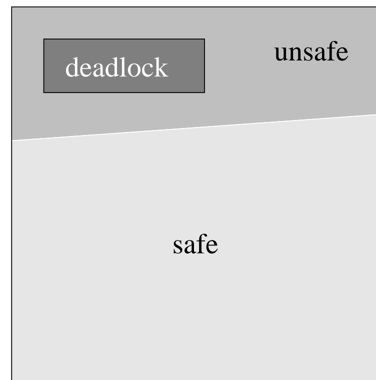


Abbildung 17: Die verschiedenen Zustände, in denen sich ein System befinden kann

besteht die zusätzliche Information darin, dass jeder Prozess zu Beginn seiner Ausführung die maximale Anzahl der Resourcen (von jedem Typ), die er während seiner gesamten Ausführung benötigt, angibt. Es ist dadurch möglich, einen Algorithmus zu erstellen, der dafür sorgt, dass das System nie in den Zustand eines Deadlocks kommt. Dazu wird der RAG dynamisch untersucht, um zu versichern, dass zyklisches Warten nicht auftreten kann.

Frage 7.10: Was ist ein „Safe State“?

Ein Zustand (**State**) ist dann sicher, wenn das System jedem Prozess in irgendeiner Reihenfolge Resourcen zuteilen kann, ohne dass es dabei zu einem Deadlock kommen kann. Ein Prozess kann dabei maximal die von ihm angegebene Anzahl an Resourcen bekommen.

Formal ausgedrückt, ist das System in einem sicheren Zustand (**Safe State**), wenn es eine sichere Sequenz (**Safe Sequence**) gibt. Eine Sequenz von Prozessen $< P_0, P_1, \ldots, P_n >$ ist eine sichere Sequenz, wenn für jeden Prozess P_i die Resourcen, welche er noch anfordern kann, mit den noch zur Verfügung stehenden Resourcen plus den Resourcen, die die Prozesse P_j, $j < i$ halten, zur Verfügung gestellt werden kann. Wenn die Resourcen für P_i nicht sofort verfügbar sein sollten, kann P_i warten, bis die P_j mit $j < i$ fertig sind und die benötigten Resourcen freigegeben werden.

Frage 7.11: Was ist der Unterschied zwischen den Zuständen „safe“, „unsafe“ und „deadlock“?

Ein System ist **unsafe**, wenn es keine sichere Sequenz gibt. Dies bedeutet jedoch noch nicht, dass es in einem Deadlock-Zustand ist. In einem unsicheren Zustand ist der zukünftige Zustand vom Verhalten der Prozesse abhängig. Es kann also sein, dass man wieder in einen sicheren Zustand gelangt, man kann aber auch in einen Deadlock geraten. Dies wird in Abbildung 17 schematisch verdeutlicht.

Ein Beispiel: Gegeben seien 12 Resourcen, z.B. Magnetbänder, und drei Prozesse P_0, P_1 und P_2, die die Resourcen wie in Tabelle 1 benutzen. Dann befindet sich das System in einem sicheren Zustand, denn die Sequenz $< P_1,\ P_0,\ P_2 >$ ist eine sichere Sequenz. Es sind noch drei Bänder unbelegt, welche P_1 nutzen kann,

Prozess	Maximal benötigt	Momentan gebraucht
P_0	10	5
P_1	4	2
P_2	9	2

Tabelle 1: Beispiel für einen sicheren Zustand (12 Resourcen insgesamt)

um seine Aufgabe zu erfüllen. Anschließend gibt der Prozess alle Resourcen ab, so dass wieder fünf Bänder zur Verfügung stehen. Also kann auch Prozess P_0 erfolgreich seine Aufgabe lösen und anschließend kann P_2 die 10 zur Verfügung stehenden Bänder benutzen, um ebenfalls fertig zu werden. Würde Prozess P_2 aus der Tabelle 1 jedoch drei Resourcen belegen anstatt zwei, dann wäre das System in einem unsicheren Zustand. Zwar könnte Prozess P_1 noch erfolgreich seine Aufgabe erledigen, jedoch ständen danach nur vier Resourcen für die beiden verbleibenden Prozesse zur Verfügung. Im schlimmsten Fall bräuchte P_0 jedoch noch fünf Resourcen und P_2 sogar sieben. Das System würde also z.B. in einen Deadlock-Zustand gelangen, wenn Prozess P_0 auf die Idee kommen würde, die restlichen fünf Resourcen anzufordern. Es könnte jedoch auch wieder ein sicherer Zustand erlangt werden, wenn z.B. P_2 im Verlauf seiner weiteren Ausführung eine Resource freigeben würde. In diesem Fall könnte Prozess P_0 seine Aufgabe wieder beenden.

Frage 7.12: In welchem Fall kann der RAG sofort benutzt werden, um Deadlock Avoidance zu betreiben?

Ist von jeder Resource nur eine Instanz verfügbar, dann kann der RAG direkt benutzt werden, um zu prüfen, ob eine Resourcenzuweisung in einen unsicheren Zustand führen würde. Dies funktioniert wie folgt: Wir fügen eine neue Klasse von gerichteten Kanten ein. Diese heißen **Claim Edges**, wobei eine gerichtete Kante $P_i \to R_j$ bedeutet, dass der Prozess P_i im Laufe seiner Ausführung irgendwann einmal auf die Resource R_j zugreifen wird. Grafisch werden Claim Edges durch gestrichelte Linien dargestellt.

Wird ein neuer Prozess ausgeführt, so werden *alle* seine Claim Edges direkt in den RAG eingefügt. Angenommen ein Prozess möchte im Laufe seiner Ausführung auf eine Resource R zugreifen. Dann wird diese Resource ihm nur dann gewährt, wenn die Umwandlung der Claim Edge in eine Assignment Edge nicht zu einem (gerichteten) Kreis führt. Entsteht ein Kreis, in dem sich mindestens eine Claim Edge befindet, so wäre das System nach Zuordnung der Resource in einem unsicheren Zustand. Würde der Kreis nur aus Assignment Edges bestehen, so wäre es ein Deadlock.

Ein Kreistest auf dem RAG mit n Prozessen benötigt einen Zeitaufwand von $O(n^2)$.

Frage 7.13: Wie kann Deadlock Avoidance in einem System, in dem Resourcen mehrere Instanzen haben können, betrieben werden?

Mit Hilfe des **Banker's Algorithmus**. Dieser Algorithmus benötigt vier Datenstrukturen. Wir gehen davon aus, dass es n Prozesse und m Resourcen gibt.

Available: Ein Array der Länge m, wobei an Position $Available[i] = k$ bedeutet, dass von Resource R_i noch k Instanzen verfügbar sind.

Max: Eine $n \times m$-Matrix (n Zeilen und m Spalten). Der Eintrag $Max[i, j] = k$ bedeutet, dass Prozess P_i maximal k Instanzen der Resource R_j braucht.

Allocation: Eine $n \times m$-Matrix. Der Eintrag $Allocation[i, j] = k$ bedeutet, dass Prozess P_i momentan k Instanzen der Resource R_j belegt.

Need: Eine $n \times m$-Matrix. Der Eintrag $Need[i, j] = k$ bedeutet, dass Prozess P_i noch k Instanzen der Resource R_j benötigt.

Wir führen folgende Ordnungsrelation ein: Seien X und Y zwei Vektoren der Länge m, dann ist $X \leq Y \Leftrightarrow X[i] \leq Y[i], \forall i \in \{1, 2, \ldots, m\}$. Im folgenden sei mit $Allocation_i$ die i-te Zeile der Allocation-Matrix bezeichnet. Entsprechendes gilt für Need.

Der Algorithmus kann nun in zwei Phasen unterteilt werden:

Safety Algorithm: Dieser Algorithmus findet heraus, ob das System sich in einem sicheren Zustand befindet, d.h. ob es irgendeine sichere Sequenz gibt. Dies funktioniert wie folgt:

1. Sei *Work* ein Vektor der Länge m und *Finish* ein Vektor der Länge n. *Work* wird gleich *Available* gesetzt und $Finish[i]$ wird auf *false* gesetzt für alle $i = 1, \ldots, n$.
2. Finde ein i, so dass gilt:
   a) $Finish[i] = false$
   b) $Need_i \leq Work$

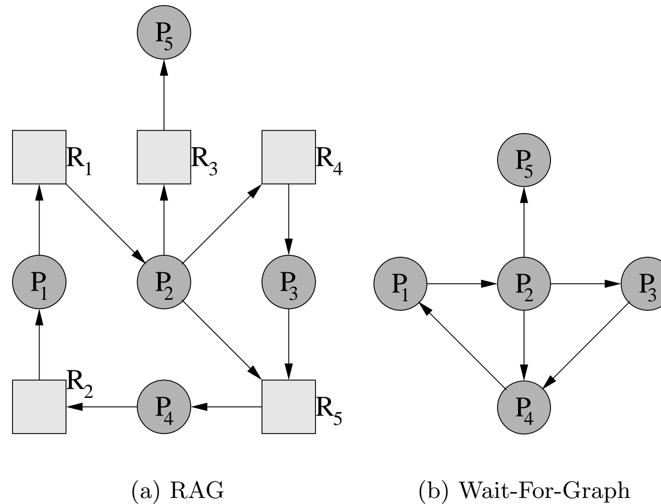
   Wenn kein solches i existiert, gehe zu 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   goto 2
4. Wenn $Finish[i] = true$ ist für alle i, dann ist das System in einem sicheren Zustand.

Die Laufzeit dieses Algorithmus beträgt $O(m \cdot n^2)$.

Resource-Request Algorithm: Sei $Request_i$ der *Request*-Vektor des Prozesses P_i. Dann bedeutet $Request_i[j] = k$, dass der Prozess P_i von der Resource R_j k Instanzen haben will. Bei jeder Anforderung von Resourcen, werden die folgenden Schritte durchgeführt:

1. Ist $Request_i \leq Need_i$? Wenn nicht, verlangt der Prozess mehr Resourcen als er zu Beginn angemeldet hat $\Rightarrow$ Fehler! Sonst gehe zu 2!
2. Ist $Request_i \leq Available$? Wenn ja, gehe zu 3, sonst muss P_i warten.
3. Tue so, als ob der Prozess P_i die Resourcen bekommt, d.h. modifiziere die Datenstrukturen wie folgt:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$



(a) RAG (b) Wait-For-Graph

Abbildung 18: Deadlock Detection mit Resourcen, die nur eine Instanz haben

Wenn der resultierende Zustand ein sicherer Zustand ist, dann gewähre dem Prozess seine Resourcen. Ansonsten muss dieser warten.

7.6 Deadlock-Detection

Frage 7.14: Wie funktioniert Deadlock Detection?
Deadlock Detection funktioniert in zwei Phasen:

- Ein Algorithmus untersucht den Zustand des Systems, um herauszufinden, ob ein Deadlock vorliegt.
- Ein Algorithmus beseitigt den Deadlock.

Frage 7.15: Wie kann beim Deadlock Detection-Ansatz einen Deadlock erkannt werden, wenn es von jeder Resource nur eine Instanz gibt?
Wir benutzen eine Variante des RAG. Der RAG wird dabei in einen **Wait-For Graph** transformiert. Zwischen zwei Prozessen P_i und P_j wird eine Kante eingeführt, wenn es eine Resource R gibt, für die gilt: $P_i \rightarrow R$ und $R \rightarrow P_j$. Gibt es in diesem Graphen einen Kreis, so liegt ein Deadlock vor. Abbildung 18(a) zeigt einen Beispielgraphen, von dem der entsprechende Wait-For-Graph in Abbildung 18(b) gezeigt ist.

Frage 7.16: Wie kann beim Deadlock Detection-Ansatz ein Deadlock erkannt werden, wenn es von jeder Resource mehrere Instanzen gibt?
Dieser Algorithmus ist dem Banker's Algorithmus sehr ähnlich. Die Datenstrukturen sind daher dieselben wie beim Banker's Algorithmus, nur dass die *Need*-Matrix nicht benötigt wird. Im Prinzip schaut der Algorithmus, ob die Prozesse noch in irgendeiner Reihenfolge fertig werden können. Er läuft wie folgt:

1. Sei *Work* ein Vektor der Länge m und *Finish* ein Vektor der Länge n. *Work* wird gleich *Available* gesetzt und $Finish[i]$ wird auf *false* gesetzt, wenn $Allocation_i \neq 0$ ist und sonst auf *true*.
2. Finde ein i, so dass gilt:
   a) $Finish[i] = false$
   b) $Request_i \leq Work$

   Wenn kein solches i existiert, gehe zu 4.
3. $Work = Work + Allocation_i$

   $Finish[i] = true$

   goto 2
4. Gibt es ein $i, 1 \leq i \leq n$ mit $Finish[i] = false$, so ist das System in einem Deadlock-Zustand. Darüber hinaus ist ein Prozess P_i an einem Deadlock beteiligt, wenn für ihn $Finish[i] = false$ gilt.

Frage 7.17: Wie oft sollte der Deadlock Detection Algorithmus ausgeführt werden?
Eine allgemeingültige Antwort kann an dieser Stelle nicht gegeben werden, vielmehr hängt die Entscheidung davon ab, wie oft Deadlocks in einem gegebenen System auftreten und wieviel Prozesse in einem solchen Fall davon betroffen sind. In einem Extrem könnte man nach jeder Resourcenanfrage eines Prozesses den Deadlock Detection Algorithmus starten, dann kann in der Menge der Prozesse derjenige Prozess ausfindig gemacht, der schlussendlich den Deadlock verursacht hat. Allerdings sorgt diese Vorgehen dafür, dass das System viel Overheadarbeit verrichten muss.

7.7 Auflösung von Deadlocks

Frage 7.18: Wie kann ein Deadlock aufgelöst werden?
Es gibt zwei Ansätze einen Deadlock aufzulösen:

Process Termination: Man kann entweder alle Prozesse, welche sich in einem Deadlock Zustand befinden, beenden oder solange sukzessiv Prozesse eines Deadlock-Kreis terminieren, bis der Deadlock aufgelöst ist. Der Vorteil der ersten Methode ist, dass der Deadlock auf jeden Fall beseitigt wird. Allerdings kann dabei eine Menge Rechenzeit verschwendet werden, wenn die Prozesse schon sehr weit in ihrer Ausführung vorangeschritten waren. Der zweite Ansatz versucht, diesen Verlust geringer zu halten. Allerdings handelt man sich nun wieder Overhead ein, da nach der Terminierung eines Prozesses ein Deadlock Detection Algorithmus ausgeführt werden muss. Die Entscheidung, welcher Prozess konkret gekillt wird, ist oftmals eine Policy-Entscheidung. Generell kann die Beendigung eines Prozesses zu Inkonsistenzen, wie z.B. inkonsistenten Dateien, führen.

Resource Preemption: Hier wird versucht, durch Freigabe von Resourcen eines Prozesses, welcher sich in einem Deadlock-Kreis befindet, diesen Deadlock-Kreis aufzulösen. Dieses Vorgehen kann natürlich sukzessiv wiederholt werden. Bei der Auswahl eines Prozesses sind einige Dinge zu beachten: Welchen Prozess wähle ich? Hier spielen Kosten eine Rolle. Was mache ich mit dem Prozess? Dadurch, dass ich dem Prozess Resourcen wegnehme, kann er nicht normal mit seiner Ausführung fortfahren. Eventuell kann man ihn in einen sicherern Zustand zurückversetzen, oder muss ihn im worst-case neu starten. Schlussendlich kann es passieren, dass ein Prozess verhungert, da ihm jedesmal, wenn ein Deadlock auftritt, die Resourcen entzogen werden. Dies sollte möglichst verhindert werden.

8 Memory Management

Frage 8.1: Zum warmwerden: Wofür benötigt man „Memory Management“?
Will man die CPU unter mehreren Prozessen aufteilen, so müssen sich diese Prozesse (zumindest teilweise) im Hauptspeicher befinden. Der Hauptspeicher muss also in irgendeiner Art und Weise aufteilt und gemanagt werden.

8.1 Einleitung

Frage 8.2: Wie sieht ein typischer „Instruction Execution Cycle“ aus?

- Instruktion aus dem Hauptspeicher holen.
- Instruktion dekodieren und evtl. Operanden aus dem Hauptspeicher holen.
- Nachdem die Instruktion ausgeführt wurde, eventuelle Ergebnisse im Hauptspeicher ablegen.

Frage 8.3: Was ist die „Input Queue“?
Die **Input Queue** ist die Menge der Prozesse, die auf der Festplatte darauf warten, in den Hauptspeicher geladen und ausgeführt zu werden.

Frage 8.4: Was ist „Binding“?
Auf dem Weg vom Source Code, über das kompilierte, ausführbare Programm, bis hin zum laufenden Prozess, werden Adressen in unterschiedlicher Art und Weise repräsentiert. Im Source Code sind diese Adressen üblicherweise in symbolischer Form (z.B. durch Variablennamen) gegeben. Diese symbolische Adresse wird nun irgendwann auf dem oben genannten Weg in eine auflösbare (physikalische) Adresse umgewandelt (z.B. 14 Bytes vom Anfang dieses Moduls).

Frage 8.5: Zu welchen Zeiten kann „Binding“ stattfinden?
Binding kann zu einem der folgenden Zeitpunkte stattfinden:

Zeitpunkt der Kompilation: Zum Zeitpunkt der Kompilation werden die symbolischen Adressen direkt in absolute Adressen umgewandelt. Dies bedeutet, dass schon zur Kompilierzeit feststehen muss, wo der Prozess später im Speicher stehen wird. Das MS-DOS `*.com`-Format benutzt absoluten Code. Nachteil: Soll das Programm in einem anderen Speicherbereich ausgeführt werden, muss es neu kompiliert werden.

Zeitpunkt des Ladens: Ist zur Kompilierzeit nicht bekannt, wo das Programm sich später im Speicher befindet, muss verschiebbarer Code generiert werden. Das Binding wird erst vollzogen, wenn das Programm geladen wird. Soll das Programm in einem anderen Speicherbereich ausgeführt werden, muss es lediglich neu geladen und nicht etwa neu kompiliert werden.

Zeitpunkt der Ausführung: Wenn der Prozess während der Ausführung im Speicher verschoben werden kann, muss das Binding bis zur Ausführung verzögert werden. Dafür ist jedoch spezielle Hardware nötig.

Frage 8.6: Was versteht man unter „Dynamic Loading“?

Bis jetzt wurde davon ausgegangen, dass sich ein Programm komplett im Speicher befinden muss, um ausgeführt zu werden. Um Speicherplatz zu sparen, kann so vorgegangen werden, dass Routinen erst in den Speicher geladen werden, wenn sie aufgerufen werden. Ruft eine Routine eine andere Routine auf, so schaut sie zuerst nach, ob sich diese im Speicher befindet. Ist dies nicht der Fall, so wird die Routine, welche sich in einem verschiebbaren Format auf einem Sekundärspeicher befindet, geladen und danach ausgeführt. Im allgemeinen braucht man für das dynamische Laden *keine* Unterstützung von Seiten des Betriebssystems.

Frage 8.7: Was versteht man unter „Dynamic Linking“?

Dynamic Linking funktioniert ähnlich wie das dynamische Laden. Viele Betriebssysteme unterstützen nur statisches Linken, d.h. die Objektmodule einer Bibliothek werden direkt in das ausführbare Programm eingebunden. Wird eine Bibliothek häufig benutzt, kann es vorkommen, dass durch statisches Linken Code unnötig dupliziert wird.

Beim dynamischen Linken werden nun nicht die Objektmodule der Bibliothek in das ausführbare Programm eingebunden, sondern sogenannte **Stubs**. Ein Stub ist ein kurzes Stück Code, welches weiß, wie eine Bibliotheksroutine aufgefunden werden kann bzw. wie diese geladen werden kann. Der Stub schaut dabei nach, ob sich die Routine schon im Speicher befindet. Ist sie nicht im Speicher, so wird sie geladen. Dabei überschreibt der Stub sich mit der Adresse der Routine, so dass man beim nächsten Aufruf der Routine Zeit spart, da diese direkt aufgerufen wird.

Dynamisches Linken muss vom Betriebssystem unterstützt werden, da das Betriebssystem die einzige Instanz ist, die prüfen kann, ob sich eine Routine evtl. im Speicherbereich eines anderen Prozesses befindet bzw. die zulassen kann, dass mehrere Prozesse auf dieselbe Speicheradresse zugreifen.

Frage 8.8: Wie funktionieren Overlays?

Die Idee bei Overlays ist es, nur die Instruktionen und Daten im Speicher zu haben, die man zu einem bestimmten Zeitpunkt benötigt. Als Beispiel wird ein 2-Pass-Assembler benutzt, der im ersten Durchgang eine Symboltabelle erstellt und im zweiten Durchgang den Maschinencode. Der Assembler wird dabei so wie in Abbildung 19 aufgeteilt.

Der Overlay Driver sorgt dafür, dass die Overlays entsprechend in den Speicher geladen werden. Zuerst wird Pass 1 geladen und ausgeführt, wobei im Anschluss in den Overlay Driver gesprungen wird, der Pass 2 lädt und ausführt.

Overlays benötigen keine Unterstützung durch das Betriebssystem, da sie durch simple Dateistrukturen realisiert werden können (lesen einer Datei und deren Ausführung). Jedoch muss der Programmierer bei der Benutzung von Overlays einige Dinge beachten, so dass Overlays sich für grosse Projekte kaum eignen und daher eher in Mikrocomputern eingesetzt werden.

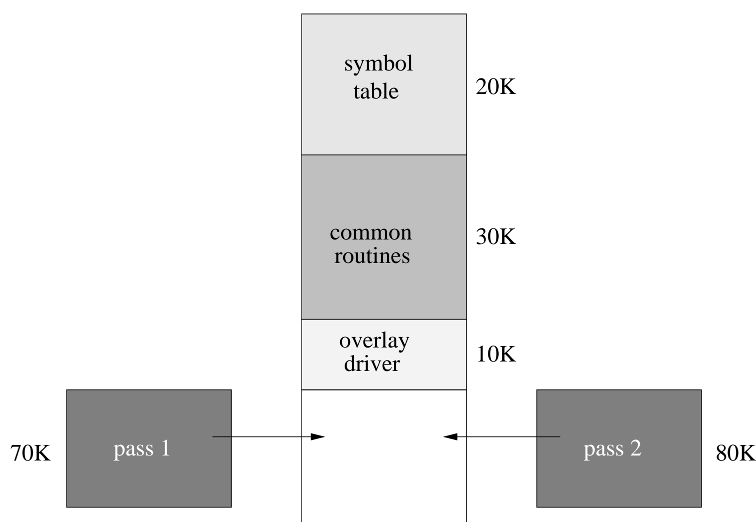


Abbildung 19: Overlays für einen 2-Pass-Assembler

8.2 Logischer und physikalischer Adressraum

Frage 8.9: Was ist der Unterschied zwischen logischen und physikalischen Adressen?

Unter **logischen Adressen** versteht man die Adressen, welche von der CPU generiert werden, während man unter **physikalischen Adressen** die Adressen vesteht, welche von der Speichereinheit benutzt werden. Beim Binding, welches zur Kompilierzeit bzw. zur Ladezeit geschieht, sind also die logischen Adressen mit den physikalischen Adressen identisch. Dies ist jedoch nicht der Fall beim Binding, welches zur Laufzeit ausgeführt wird. In diesem Fall werden die logischen Adressen auch **virtuelle Adressen** genannt.

Die Abbildung vom logischen auf den physikalischen Speicher wird von einem sogenannten **Memory Management Unit (MMU)** ausgeführt. Dabei gibt es verschiedene Verfahren, wie diese Abbildung bewerkstelligt werden kann, wovon einige noch in diesem Text besprochen werden.

Es ist wichtig festzuhalten, dass die CPU (bzw. ein Prozess) nie physikalische Adressen sieht, sondern immer nur auf logischen Adressen arbeitet. So kann die CPU z.B. einen Zeiger auf die (logische) Adresse 346 erzeugen und damit arbeiten. Die Adresse wird erst aufgelöst, wenn wirklich auf den Speicher zugegriffen werden muss.

Frage 8.10: Gebe ein einfaches Verfahren zur Relokierung von Speicher an!

Das simpelste Verfahren ist die Verwendung eines sog. **Relocation Registers**. Der Wert, der in diesem Register steht, wird einfach auf jede logische Adresse aufaddiert und ergibt so die physikalische Adresse. Steht im Register z.B. der Wert 14000, so wird aus der logischen Adresse 346 die physikalische Adresse 14346. Ist R der Wert des Relocation Registers, so geht der logische Adressraum von 0 bis max und der physikalische von $R + 0$ bis $R + max$. Dieses Verfahren ist in Abbildung 20 dargestellt.

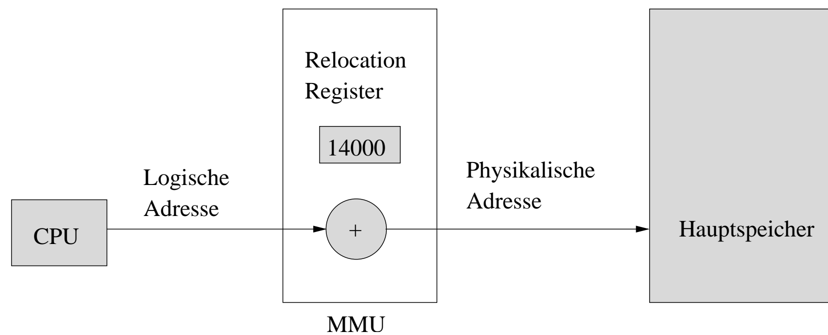


Abbildung 20: Relokierung mit einem Relocation Register

8.3 Swapping

Frage 8.11: Was ist „Swapping"?
Beim **Swapping** wird ein Prozess auf einen **Backing Store** geschrieben, um im Speicher mehr Platz, z.B. für einen anderen Prozess zu schaffen. An einem späteren Zeitpunkt wird er dann vom Backing Store wieder in den Hauptspeicher geladen.

Diese Technik kann zum Beispiel beim Round-Robin CPU Scheduling benutzt werden. Hat ein Prozess seinen Zeitschlitz verbraucht, wird er aus dem Speicher rausgeswappt und ein neuer Prozess reingeswappt. Dabei sollte das rausswappen allerdings nicht länger dauern als der Zeitschlitz groß ist, da dieses Vorgehen sonst sehr ineffizient wird (es ist so oder so nicht sehr effizient).

Frage 8.12: Worauf muss beim „Swapping" geachtet werden?
Man muss darauf achten, dass der Prozess komplett unbeschäftigt (**idle**) ist und nicht z.B. auf die Beendigung einer Ein-/Ausgabe wartet. Denn ansonsten kann es bei asynchroner I/O passieren, dass ein Gerät in den Speicher eines Prozesses schreibt, der sich nun an der Stelle befindet, an der sich vorher der Prozess befunden hat, der die I/O angestoßen hat. Es gibt zwei Möglichkeiten dies zu umgehen:

1. Prozesse, die auf I/O warten, werden nicht geswappt.
2. Es werden nur Betriebssystem-Puffer für I/O benutzt.

8.4 Zusammenhängende Allocation

Frage 8.13: Was versteht man unter „Contiguous Allocation" (zusammenhängende Allokation)?
Grob kann der Hauptspeicher in zwei Bereiche eingeteilt werden. Einer, in dem sich das Betriebssystem befindet und einer, in dem sich die Userprogramme befinden. Angenommen, es befindet sich immer nur das Betriebssystem und *ein* Userprozess im Speicher. Dann braucht man nur ein Limit und ein Relocation Register um Speicherschutz zu gewährleisten. Im Limitregister steht die maximale logische Adresse. Findet ein Speicherzugriff auf eine logische Adresse, die

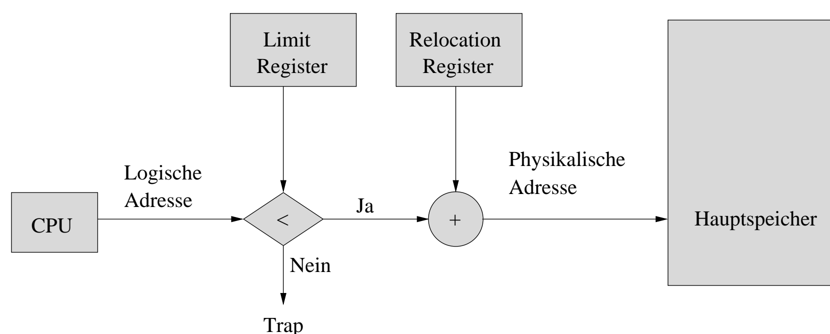


Abbildung 21: Relokierung mit einem Limit und einem Relocation Register

kleiner als dieser Wert ist, statt, wird der Inhalt des Relocation Register wie gehabt dazu addiert. Durch diese beiden Register kann sich der Platz, den das Betriebssystem beansprucht auch ruhig ändern, da (bei relokierbarem Code) nur die beiden Register entsprechend angepasst werden müssen. Abbildung 21 verdeutlicht dieses Schema.

Frage 8.14: Wie funktioniert zusammenhängende Allokation, wenn sich mehrere Prozesse im Hauptspeicher befinden?

Die zusammenhängende Allokation mit mehreren Prozessen funktioniert im Prinzip genauso wie mit einem Prozess, nur dass sich das Betriebssystem in einer Tabelle merken muss, welche Speicherbereiche noch frei sind. Wir gehen davon aus, dass zu Beginn ein einziger großer Speicherbereich frei ist, ein **Loch**. Kommt nun ein neuer Prozess an, wird geschaut, ob es ein Loch gibt, welches groß genug ist, so dass er vollständig hineinpasst. Ist dies der Fall, bekommt der Prozess den Platz und der restliche freie Speicher wird für zukünftige Anfragen von weiteren Prozessen benutzt. Wird ein Prozess beendet, wird der Platz wieder frei.

Gibt es kein Loch, welches groß genug ist für einen Prozess, so kann entweder gewartet werden, bis eines frei wird oder es wird nachgeguckt, ob ein anderer Prozess den verfügbaren Platz eventuell nutzen kann.

Nach einer Zeit gibt es mehrere Löcher im System, also eine Menge von Löchern (**set of holes**), die für Prozesse benutzt werden können.

Frage 8.15: Wie kann entschieden werden, in welches Loch ein Prozess gepackt wird?

Hier gibt es drei mögliche Verfahren:

First-Fit: Es wird das erste Loch genommen, welches groß genug ist. Die Suche wird entweder am Anfang der Menge der Löcher gestartet oder dort, wo bei der letzten Suche aufgehört wurde.

Best-Fit: Es wird das Loch gesucht, in das der Prozess am besten hineinpasst. Also dasjenige, das das kleiste verbleibende Loch produziert. Ist die Menge der verfügbaren Löcher nicht der Größe nach geordnet, so müssen alle Löcher betrachtet werden.

Worst-Fit: Es wird das größte Loch genommen, also dasjenige, welches das größte verbleibende Loch produziert.

Simulationen zeigen, dass First-Fit und Best-Fit anscheinend bessere Strategien sind als Worst-Fit.

Frage 8.16: Was ist externe bzw. interne Fragmentierung?

Wird ein Prozess beendet, so wird sein Speicher wieder freigegeben. Nach einer Zeit kann es dazu kommen, dass relativ viele kleine Löcher im Speicherbereich existieren. **Externe Fragmentierung** liegt vor, wenn theoretisch genug Speicher für eine Prozessanfrage zur Verfügung stehen würde, dieser jedoch nicht zusammenhängend ist.

Im Zusammenhang mit Fragmentierung gibt es die sog. 50%-Regel, die besagt, dass bei der Strategie First-Fit für N allokierte Blöcke ca. $0.5N$ Blöcke durch Fragmentierung verloren gehen, d.h. ein Drittel des Speichers unbrauchbar ist.

Während bei der externen Fragmentierung Platz zwischen den Partitionen eventuell verschenkt wird, wird bei der **internen Fragmentierung** Platz innerhalb einer Partition verschenkt, d.h. es wird für einen Prozess mehr Speicher allokiert als er eigentlich bräuchte. Dies kann jedoch unter Umständen sinnvoll sein. Angenommen wir haben ein Loch dessen Größe 20000 Bytes beträgt und einen Prozess, der 19998 Bytes benötigt. Wenn wir diesem Prozess nun genau den Speicher geben, den er benötigt, so bleibt ein Loch von 2 Bytes übrig. Der Overhead zur Verwaltung eines Lochs dieser Größe dürfte jedoch größer sein als das Loch selbst, so dass es besser wäre, dem Prozess einfach die kompletten 20000 Bytes zu geben.

Frage 8.17: Wie kann das Problem der externen Fragmentierung gelöst werden?

Das Problem der externen Fragmentierung kann mit Hilfe von **Compaction** gelöst werden. Bei Compaction werden die Prozesse so im Speicher verschoben, dass wieder ein grosses Loch entsteht. Dabei gibt es mehrere Möglichkeiten, wie dies gemacht werden kann. Man kann z.B. alle Prozesse in Richtung Anfang oder Ende des Speichers verschieben, was jedoch sehr teuer sein kann. Es kann aber auch vorkommen, dass man nur einen Prozess in ein Loch im Speicher verschieben muss, damit wieder ein grosses Loch entsteht.

8.5 Paging

Frage 8.18: Was ist „Paging“?

Werden Prozesse zusammenhängend im Speicher gehalten, so müssen sie, wenn sie rausgeswappt werden, auch zusammenhängend auf den Backing Store geschrieben werden. Dabei ergibt sich das Problem der Fragmentierung auch auf dem Backing Store, nur dass hier aus zeitlichen Gründen keine Compaction durchgeführt werden kann. Die Grundidee des **Paging** ist es nun, zu erlauben, dass der physikalische Adressraum eines Prozesses sowohl im Hauptspeicher als auch auf dem Backing Store nicht zusammenhängend ist. Dies bedeutet, dass der Prozess Speicher zugewiesen bekommt, wo immer dieser gerade frei ist.

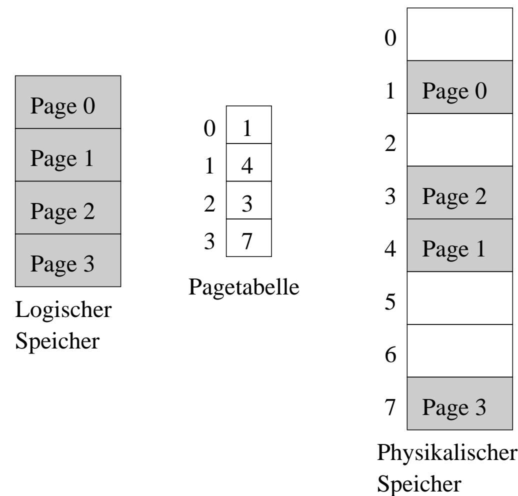


Abbildung 22: Modell des logischen und physikalischen Speichers beim Paging

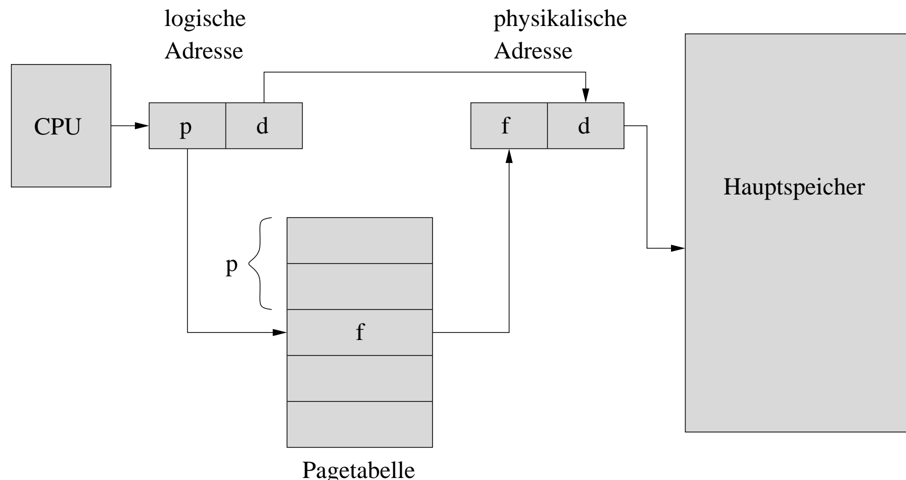


Abbildung 23: Grundlegende Idee des Paging

Der physikalische Speicher wird dabei in in Blöcke fester Größe aufgeteilt. Diese Blöcke werden auch **Frames** genannt. Der logische Speicher wird in Blöcke der gleichen Größe zerteilt, welche **Pages** genannt werden. Ebenso wird der Platz auf dem Backing Store in solche Blöcke aufgeteilt. Mit Hilfe einer Tabelle, der sogenannten Pagetabelle (Seitentabelle) werden die logischen Pages den physikalischen Frames zugeordnet.

Frage 8.19: Wie wird eine logische Adresse beim „Paging“ interpretiert?
Beim Paging wird eine logische Adresse in zwei Teile aufgeteilt: Die Pagenummer und der Offset in diese Page. Die Pagenummer wird als Index in eine Pagetabelle benutzt, in der die Basisadresse im physikalischen Speicher für diese Page steht. Zu dieser Basisadresse wird dann noch der Offset hinzuaddiert und ergibt so die endgültige physikalische Adresse. Diese Grundidee ist in Abbildung 23 dargestellt.

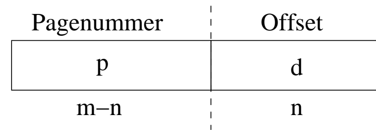


Abbildung 24: Die Aufteilung einer logischen Adresse beim Paging

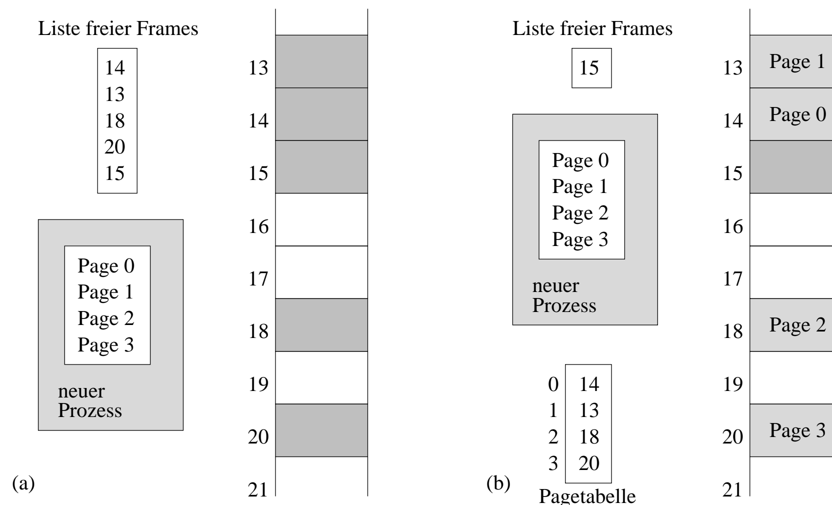


Abbildung 25: Verwaltung und Zuweisung freier Frames

Die Größe einer Page ist in den meisten Fällen eine Zweierpotenz. Der Grund hierfür ist der, dass in diesem Fall die Umrechnung einer logischen Adresse in eine physikalische Adresse besonders einfach ist. Beträgt die Größe des logischen Adresseraumes 2^m und die Größe einer Page 2^n adressierbare Einheiten, so gibt es 2^{m-n} verschiedene Pages. Also geben die ersten $m - n$ Bit der logischen Adresse die Page an und die restlichen n Bit den Offset in diese Seite (siehe Abbildung 24).

Wenn ein Prozess also in den Hauptspeicher geladen wird, wird seine Größe in Pages gemessen und dementsprechend viele Pages allokiert. Dazu müssen also die freien Frames des Systems verwaltet werden.

Frage 8.20: Wie sieht es beim „Paging“ mit externer bzw. interner Fragmentierung aus?

Externe Fragmentierung gibt es nicht, da jeder freie Frame benutzt werden kann. Da Prozesse jedoch nicht immer ein Vielfaches der Pagegröße groß sind, kann es zu interner Fragmentierung kommen. Im Extremfall kann es passieren, dass ein Prozess für ein einziges Byte eine Extrapage anfangen muss. Im Schnitt wird eine halbe Page vergeudet. Daher könnte man annehmen, dass ein kleine Pagegröße vorteilhaft ist, jedoch bedeuten kleinere Pages auch mehr Verwaltungsaufwand.

Frage 8.21: Wie werden Page Tables über Hardware unterstützt.

Hier werden zwei Möglichkeiten beschrieben: **dedizierte Register** und **Page Table Base Register (PTBR)** mit **Translation Look-Aside Buffern**

(TLB).

Eine Möglichkeit ist es, die Pagetabelle in dedizierten Registern zu speichern, so dass schnell auf diese zugegriffen werden kann. Diese Möglichkeit ist jedoch nur effizient anwendbar, wenn die Größe der Pagetabelle nicht allzu groß ist (z.B. bis 256 Einträge). Bei modernen Rechnern kann die Pagetabelle jedoch sehr groß werden (z.B. eine Million Einträge).

Bei der Verwendung eines Page Table Base Registers wird die Pagetabelle im Hauptspeicher gespeichert und die Startadresse der Tabelle im PTBR gespeichert. Bei einem Context Switch wird sich natürlich auch der Inhalt des PTBR gemerkt. Der Nachteil dieses Vorgehens ist, dass zwei Speicherzugriffe notwendig werden, um ein Datum (z.B. einen Operanden) aus dem Speicher zu lesen. Zuerst muss die Pagenummer mit dem Inhalt des PTBR addiert werden, damit man weiss, wo sich der Pagetabellen-Eintrag im Speicher befindet. Anschließend muss mit dem Inhalt des Pagetabellen-Eintrags und dem Offset der Inhalt des Speichers ausgelesen werden. Damit wird der Speicherzugriff um den Faktor zwei verlangsamt, was nicht zumutbar ist.

Abhilfe zu diesem Problem bietet ein **assoziatives Register**, welches auch TLB genannt wird. Assioziative Register sind wie folgt aufgebaut. Jedes Register besteht aus zwei Teilen: einem Schlüssel und einem zum Schlüssel gehörenden Wert. Werden die assoziativen Register nun mit einem Schlüssel „gefüttert“, vergleichen alle Register gleichzeitig ihren Schlüssel mit dem gegebenen. Wird der Schlüssel gefunden, wird der Wert ausgegeben. In unserem Fall sind die Schlüssel der assoziativen Register die Pagenummern und die dazugehörigen Werte die entsprechenden Framenummern.

Soll eine logische Adresse also umgewandelt werden, gibt man die Pagenummer an das TLB. Befindet sich die Pagenummer im TLB bekommt man sofort die Framenummer zurück und wie üblich wird mit Hilfe des Offset der Speicherinhalt ausgelesen. Ansonsten müssen wir im Hauptspeicher nachschauen und den Wert von dort holen. Dabei schreiben wir die Pagenummer und die entsprechenden Framenummer in den TLB.

Frage 8.22: Was muss man bei der Verwendung von TLBs beachten?

Bei einem Context Switch muss der TLB geflusht werden, da ansonsten die logischen Adressen des neuen Prozesses mit falschen physischen Adressen assoziiert sein können.

Frage 8.23: Was bezeichnet die „Hit Ratio“ bei TLBs?

Die **Hit Ratio** bezeichnet den Anteil der Zugriffe in Prozent, bei denen eine Pagenummer im TLB gefunden wird. Ist die Zugriffszeit des TLBs und die Zugriffszeit auf den Hauptspeicher gegeben, so kann mit Hilfe der Hit Ratio die **effektive Speicherzugriffszeit** ermittelt werden. Sei t die Zeit, die ein TLB-Zugriff benötigt und s die Zeit für einen Speicherzugriff. Dann errechnet sich die effektive Speicherzugriffszeit e wie folgt:

$$\begin{aligned} e &= \alpha(t+s) + (1-\alpha)(t+2s) \\ &= t + (2-\alpha)s \end{aligned}$$

Dabei ist α die Wahrscheinlichkeit dass eine Page im TLB gefunden wird. Ist $\alpha = 1$ (wir finden die Pagenummer immer im TLB), ergibt die Formel $e = t+s$,

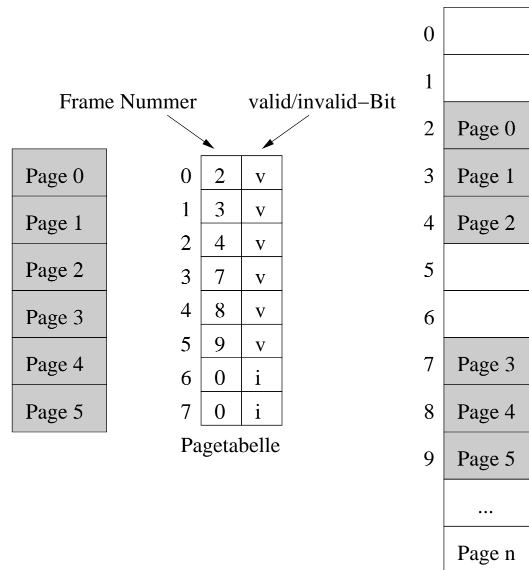


Abbildung 26: Grundlegende Idee des Paging

d.h. wir benötigen die Zeit für das Nachschauen im TLB und den eigentlichen Speicherzugriff. Ist hingegen $\alpha = 0$ (der TLB bringt quasi nichts), so ergibt sich $e = t + 2s$, d.h. wir schauen im TLB nach, finden nichts und müssen doch zwei Speicherzugriffe machen.

Frage 8.24: Welche Möglichkeiten gibt es, den Speicher beim Paging zu schützen?

Eine Möglichkeit ist es, für jeden Eintrag in der Pagetabelle zusätzliche Bits zu spendieren. So kann man z.B. ein zusätzliches Bit benutzen, um festzustellen, ob eine Page nur lesbar oder les- und beschreibbar ist. Wird versucht, auf eine nur lesbare Page zu schreiben, gibt es einen Trap, der durch die Hardware ausgelöst wird, so dass das Betriebssystem die Kontrolle erlangt.

Weiterhin ist es möglich, ein **valid/invalid**-Bit zu benutzen, welches angibt, ob eine Page noch im logischen Adressraum eines Prozesses liegt. Dies wird z.B. benutzt, wenn ein Prozess stets eine Pagetabelle besitzt, die den ganzen logischen Adressraum abdeckt, der Prozess jedoch nicht den ganzen Raum benutzt.

Da Prozesse oft nur einen Bruchteil des logischen Raumes nutzen und dementsprechend auch nicht eine komplette Pagetabelle benötigen, die nur wertvollen Speicherplatz verschwenden würde, benutzt man oft ein **Page Table Length Register (PTLR)**, welches die Länge der Pagetabelle speichert.

Frage 8.25: Warum wird „Multilevel Paging“ benutzt? Wie funktioniert es?

Moderne Computersysteme unterstützen einen sehr großen logischen Adressraum (z.B. 2^{32} oder 2^{64}), so dass die Pagetabelle selbst sehr groß wird. Bei einem logischen Adressraum, der 32 Bit breit ist, benötigt man bei einer Pagegröße von 4 K (2^{12}) Speichereinheiten (Byte, Wort, etc.) $\frac{2^{32}}{2^{12}} = 2^{20} = 1048576$ Einträge in die Pagetabelle. Jeder Eintrag ist 4 Byte gross, also belegt nur die Pagetabelle

Pagenummer		Offset
p_1	p_2	d
10	10	12

Abbildung 27: Eine gepagete Pagetabelle

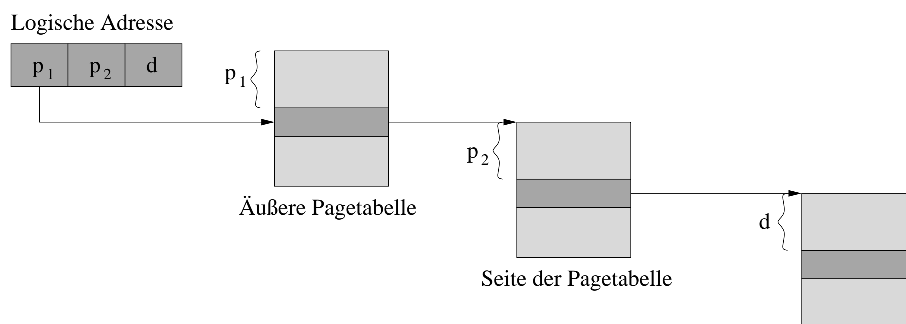


Abbildung 28: Grundlegende Idee des Paging

eines Prozesses maximal 4 MB. Da die Pagetabelle so groß werden kann, ist es also nicht erwünscht, dass sie an einem Stück im Speicher liegt (wenn wir das effizient könnten, würden wir uns ja nicht mit dem Paging auf den Kopf stellen). Das Problem das wir etwas nicht an einem Stück im Speicher liegen haben wollen, haben wir schon einmal gelöst: mit Hilfe des Paging. Genau dieselbe Lösung wird nun hier verwendet. Die Pagetabelle wird gepagt.

Dabei wird die logische Adresse einfach noch weiter aufgeteilt. Angenommen wir haben eine 32 Bit Maschine, dann kann die logische Adresse z.B. so wie in Abbildung 27 aufgeteilt werden.

Die ersten 10 Bit p_1 werden benutzt, um in eine äußere Pagetabelle zu indizieren. Dort finden wir an Stelle p_1 die Adresse, an der ein Teil der gepageten eigentlichen Pagetabelle anfängt. In diese indizieren wir mit p_2 und bekommen nun die Framenummer, welche in Kombination mit dem Offset zum Speicherinhalt führt. Das Prinzip wird in Abbildung 28 noch einmal verdeutlicht.

Wie man sieht, sind hier nun noch mehr Speicherzugriffe nötig. Bei einer 64-Bit-Architektur können es noch mehr werden, da die äußere Pagetabelle eventuell noch einmal gepaget werden muss. Abhilfe schafft jedoch auch hier wieder ein assoziatives Register (TLB).

Frage 8.26: Was ist eine invertierte Pagetabelle?

Bis jetzt hatte jeder Prozess seine eigene Pagetabelle, die unter Umständen sehr groß werden konnte. Falls jeder Prozess eine Pagetabelle für seinen kompletten logischen Adressraum hat, wird unter Umständen auch sehr viel Platz verschwendet. Invertierte Pagetabellen gehen anders vor. Bei diesem System gibt es nur eine große Pagetabelle, die einen Eintrag für jeden physikalischen Frame besitzt. Ein virtuelle Adresse bei diesem System sieht wie folgt aus:

```
<Prozess-id, Pagenummer, Offset>
```

Zur Suche in der invertierten Pagetabelle wird nur das Paar `Prozess-id` und

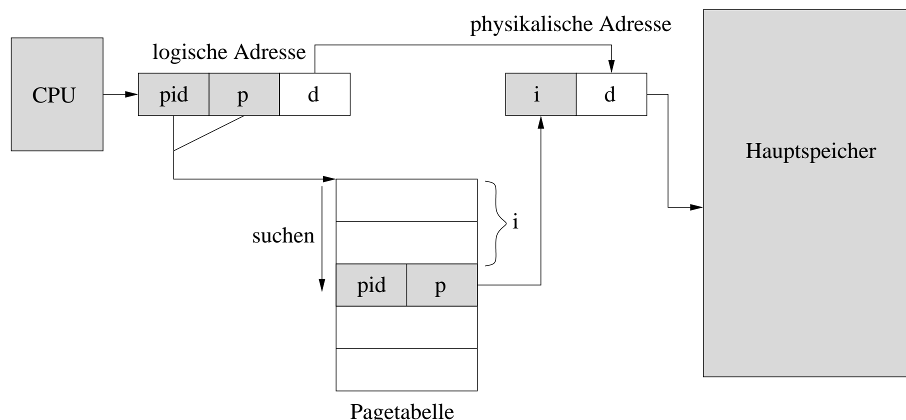


Abbildung 29: Prinzip einer invertierten Pagetabelle

`Pagenummer` benötigt. Die Tabelle wird so lange durchlaufen, bis dieses Paar gefunden wird. Aus dem Abstand i, den man in die Tabelle hineinläuft, um das Paar zu finden, lässt sich errechnen, wo im physikalischen Speicher der Frame anfängt. Zu dieser Startadresse wird noch der Offset hinzugerechnet und schon hat man das gesuchte Datum. Dieses Vorgehen wird in Abbildung 29 graphisch dargestellt.

Zwar ist die Pagetabelle nun nicht mehr so groß bzw. ihre Größe ist beschränkt, jedoch muss man dafür in der Tabelle suchen, was sehr lange dauern kann. Daher ist es üblich eine Hashtabelle zu benutzen. Desweiteren kann auch hier wieder einiges mit Hilfe von assoziativen Registern beschleunigt werden.

Frage 8.27: Wie können Pages zwischen Prozessen geteilt werden?

Ein Vorteil des Paging ist der, dass oft benutzter Code geteilt werden kann. Angenommen, auf einem System, welches 40 Benutzer hat, würden alle Benutzer gleichzeitig einen Texteditor verwenden, der zu 150K aus Code besteht und zu 50K aus einer Datensektion. Dann würden insgesamt 8000K benötigt. Es ist jedoch auch möglich, dass die Codesektion des Texteditors nur einmal im Speicher liegt und nur noch 40 Datensektionen benötigt werden. Dadurch würde eine Menge Speicherplatz gespart werden. Die Pages, in denen sich die Codesektion des Programmes befindet, würden also geteilt werden.

Damit dies funktioniert, muss der Code des Texteditors jedoch **reentrant** sein, was bedeutet, dass sich der Code z.B. nicht selbst modifiziert. Ansonsten würde es Probleme geben, wenn mehrere Programme nebenläufig auf diesen Code zugreifen würden. Eine Methode dies zu erreichen, ist es, Daten auf denen Funktionen bzw. Methoden arbeiten, nur als Zeiger zu übergeben. Desweiteren ist es hilfreich, wenn das Betriebssystem erzwingt, dass die Codesektion des Editors, im obigen Beispiel, nur lesbar und nicht beschreibbar ist.

Desweiteren sei angemerkt, dass das Teilen von Pages bei Systemen, die invertierte Pagetabellen benutzen, nur sehr schwer zu realisieren ist. Denn zu jedem physikalischen Frame gibt es nur eine korrespondierende virtuelle Adresse in der invertierten Pagetabelle. Bei nicht invertierten Pagetabellen, wenn also jeder Prozess seine eigene Tabelle besitzt, gibt es einfach zu zwei virtuellen

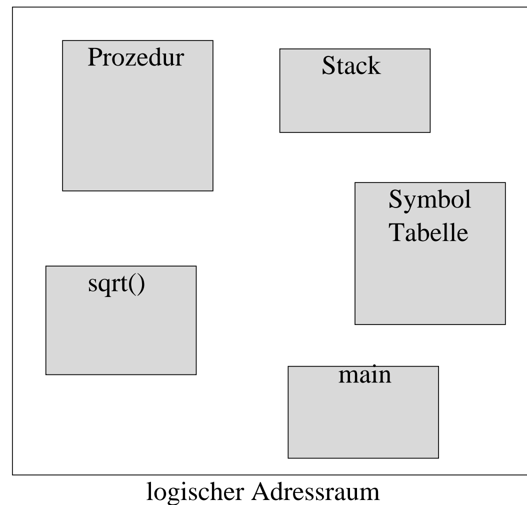


Abbildung 30: Benutzerperspektive auf ein Programm

Adressen dieselbe physikalische Adresse in den jeweiligen Pagetabellen.

8.6 Segmentierung

Frage 8.28: Was ist die Grundidee bei der Segmentierung („Segmentation")?

Benutzer betrachten den logischen Adressraum im allgemeinen nicht als ein lineares Array von Speicherstellen, in dem sich an einigen Stellen Code befindet und an wieder anderen Daten. Vielmehr betrachten Benutzer den Speicher als eine Ansammlung von Segmenten, die sich irgendwo im Speicher befinden. Ein Programm kann zum Beispiel in die Segmente `main`-Funktion, Arrays, Stacks, Methoden, etc. zerlegt werden, wobei es dem Benutzer egal ist, ob ein bestimmtes Segment vor oder hinter einem anderen Segment liegt (Abbildung 30).

Segmentierung ist nun ein Speichermanagement-Verfahren, welches diese Sichtweise des Benutzers unterstützt. Der logische Adressraum ist dabei eine Ansammlung von Segmenten, wobei jedes Segment einen Namen und eine Länge hat. Durch einen Offset können Elemente (z.B. Variablen oder Instruktionen) innerhalb eines Segmentes referenziert werden (z.B. die Integervariable `a` steht 23 Bytes vom Anfang des Segmentes entfernt). Da Computer besser mit Zahlen als mit Namen umgehen können, wird ein Segment intern nicht durch einen Namen sondern durch eine Zahl referenziert. Somit besteht eine logische Adresse aus dem Tupel:

`<Segment-Nummer, Offset>`

Segmente werden im allgemeinen durch den Compiler erstellt, der zum Beispiel ein Segment für lokale Variablen, eines für die `main`-Funktion, etc. erstellt. Der Loader weist diesen einzelnen Segmenten dann während des Ladevorgangs Segmentnummern zu.

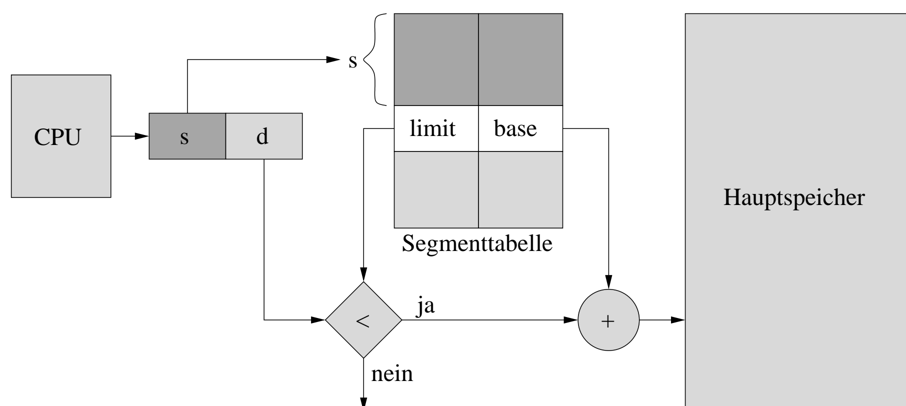


Abbildung 31: Abbildung der logischen auf die physikalischen Adressen beim Segmentieren

Frage 8.29: Wie werden bei der Segmentierung die logischen Adressen auf den physikalischen Speicher abgebildet?

Dazu wird eine sogenannte **Segmenttabelle** benutzt. Jeder Eintrag in dieser Tabelle besteht aus einer **Segment Base** und einem **Segment Limit**. Die Segment Base gibt an, wo das Segment im physikalischen Speicher steht und das Segment Limit, wie lang es ist.

Die Abbildung funktioniert nun wie folgt, wobei davon ausgegangen wird, dass die logische Adresse als die Segmentnummer s und einem Offset d gegeben ist. Wir nehmen die Segmentnummer s und schauen in der Segmenttabelle an Position s nach. Ist der Offset d kleiner als das Segment Limit, lesen wir den Segment Base-Wert aus und addieren den Offset dazu, um die physikalsiche Adresse zu erhalten. Ist der Offset größer als das Segment Limits, wird ein Trap ausgelöst, der dem Betriebssystem die Kontrolle übergibt.

Frage 8.30: Wie funktioniert „Sharing“ bei der Segmentierung? Welche Probleme kann es geben?

Bei der Segmentierung können Segmente geteilt werden. Segmente werden geteilt, wenn zwei Einträge in den Segmenttabellen verschiedener Prozesse auf dieselbe physikalische Adresse zeigen. Probleme kann es geben, wenn Segmente auf sich selbst referenzieren. Wenn ein geteiltes Segment in der Segmenttabelle des einen Prozesses die Nummer 4 hat und in der Tabelle eines anderen Prozesses die Nummer 7, wie soll es dann auf sich selbst referenzieren? Gelöst werden kann dieses Problem mit Hilfe von relativen Operationen, wie z.B. der Verwendung von relativen Sprüngen anstatt von absoluten.

Frage 8.31: Wie funktioniert „Segmentation mit Paging“?

Bei dem bis hier beschriebenen Verfahren der Segmentierung musste ein einzelnes Segment vollständig im Speicher liegen. Bei großen Segmenten kann dies wieder zu starker externer Fragmentierung führen. Die Lösung zu diesem Problem liegt quasi auf der Hand. Die einzelnen Segment werden wiederum gepaget und können so überall abgelegt werden, wo gerade Platz ist.

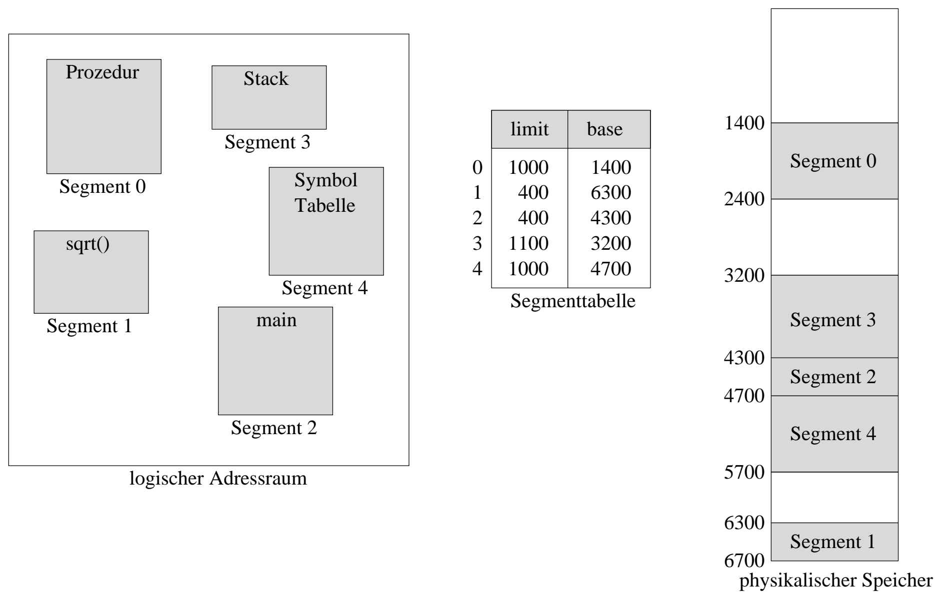


Abbildung 32: Beispiel für Segmentierung

9 Virtual Memory

9.1 Einleitung

Frage 9.1: Was versteht man unter „Virtual Memory“?
Virtual Memory ist ein Verfahren, welches es erlaubt, Prozesse auch dann auszuführen, wenn sich diese nicht vollständig im Speicher befinden. Es wird in den meisten Fällen mit Hilfe von **Demand Paging** realisiert.

Frage 9.2: Welche Vorteile bietet „Virtual Memory“?
Virtual Memory bietet unter anderem die folgenden Vorteile:

- Programme können größer als der Hauptspeicher sein. Für Programmierer bedeutet dies, dass sie ihre Programme für einen sehr grossen Adressraum schreiben können.
- Da Programme nun unter Umständen weniger Platz brauchen, kann der Grad des Multiprogramming erhöht werden und mehr Programme in den Hauptspeicher geladen werden, was zu erhöhter CPU-Ausnutzung und erhöhtem Durchsatz führt.
- Es wird weniger I/O benötigt.

9.2 Demand Paging

Frage 9.3: Wie funktioniert „Demand Paging“?
Wenn ein Prozess reingeswappt werden soll, schätzt der Pager ab, welche Pages wahrscheinlich benutzt werden, bevor der Prozess wieder herausgeswappt wird.

Der Pager lädt dann auch nur diese Pages in den Hauptspeicher, anstatt den ganzen Prozess zu laden.

Frage 9.4: Was ist ein „Lazy Pager“? Was ist der Unterschied zwischen einem Pager und einem Swapper?

Ein **Lazy Pager** lädt eine Page erst in den Speicher, wenn diese zum ersten Mal benötigt wird. Ein Pager swappt Pages in den Speicher bzw. auf die Platte, während ein Swapper ganze Prozesse swappt.

Frage 9.5: Wie kann unterschieden werden, ob sich eine Page im Speicher befindet oder nicht?

Hier kann Hardwaresupport in Form eines **valid/invalid**-Bits benutzt werden. Seiten, welche sich im Speicher befinden werden als *valid* bezeichnet, während Seiten die sich nicht im Speicher bzw. im logischen Adressraum des Prozesses befinden als *invalid* markiert werden.

Frage 9.6: Was passiert, wenn sich eine Page nicht im Speicher befindet?

Befindet sich eine Page nicht im Speicher ist sie als *invalid* markiert. Das Lesen einer Seite, die *invaild* ist, verursacht einen **Page-Fault Trap** zu Betriebssystem. Dies führt nun die folgenden Schritte aus:

1. In einer internen Tabelle wird nachgeschaut, ob der Speicherzugriff erlaubt war.
2. War der Speicherzugriff nicht erlaubt, wird der entsprechende Prozess beendet. War er erlaubt, wird die entsprechende Seite nun in den Speicher geladen.
3. Es wird ein leerer Frame gesucht.
4. Es wird eine Plattenoperation angestoßen, die die Seite in den Frame liest.
5. Ist der Lesevorgang beendet, wird die zum Prozess gehörige Tabelle und die Pagetabelle aktualisiert.
6. Der Prozess wird mit der Instruktion, die den Page Fault verursacht hat, neu gestartet.

Dabei wird der Status eines Prozesses bei der Bearbeitung des Page Faults natürlich gespeichert, damit er an exakt derselben Stelle fortgesetzt werden kann, nur dass die Seite sich dann im Speicher befindet.

Frage 9.7: Was ist „Pure Demand Paging“?

Beim **Pure Demand Paging** wird eine Seite so lange nicht in den Speicher geladen, bis sie gebraucht wird. Dies bedeutet, dass ein Prozess zu Beginn seiner Ausführung keine Page im Speicher hat und dementsprechend mit großer Wahrscheinlichkeit zu Beginn viele Page Faults verursacht.

Frage 9.8: Welche Hardware wird für „Demand Paging“ benötigt?

Im Grunde genommen dieselbe wie für das normale Paging und Swapping:

- Pagetabelle, die *valid/invalid*-Bits oder einen ähnlichen Mechanismus besitzt.

- Sekundärspeicher, auf dem sich die Seiten befinden, welche nicht im Hauptspeicher sind.

Frage 9.9: Warum können „Page Faults“ eventuell ziemlich häufig auftreten?
Es kann z.B. passieren, dass das Holen der nächsten Instruktion einen Page Fault auslöst. Ist die Seite geladen und die Instruktion dekodiert, kann es passieren, dass die Operanden erneut einen Page Fault verursachen. Wenn die Operanden indirekter Natur sind, kann ein geladener Operand noch einen Page Fault auslösen.

Desweiteren gibt es Architekturen, welche z.B. mehrere Bytes im Speicher von einer Stelle zu einer anderen bewegen können. Überqueren die Daten, die bewegt werden sollen, eine Pagegrenze kann das ganze recht unangenehm werden.

9.3 Performanz von Demand Paging

Frage 9.10: Wie kann die effektive Zugriffszeit für Demand Paging berechnet werden?
Sei p die Wahrscheinlichkeit, dass ein Page Fault auftritt. Dann lässt sich die effektive Zugriffszeit e wie folgt berechnen, wobei ma die Zeit für einen Speicherzugriff auf eine Seite, die sich im Speicher befindet, ist:

$$e = (1 - p) \cdot ma + p \cdot \text{page fault time}$$

Die Page Fault-Zeit ergibt sich dabei grob aus den folgenden Schritten:

1. Ausführen des Codes für den Page Fault-Interrupt.
2. Lesen der Seite.
3. Restarten des Prozesses.

Dabei benötigt Schritt 2 die meiste Zeit. Diese Zeit kann unter Umständen ein wenig reduziert werden, wenn ein Prozess, bevor er ausgeführt wird, erst in den Swap Space kopiert wird, da der Zugriff auf diesen Speicher im allgemeinen schneller ist, als der Zugriff über das reguläre Dateisystem.

9.4 Seitenersetzung (Page Replacement)

Frage 9.11: Wie kann vorgegangen werden, wenn es keine freien Frames mehr gibt?
Wenn mehrere Prozesse laufen, kann es passieren, dass es irgendwann keine freien Frames mehr gibt, da der komplette Speicher schon benutzt wird. In diesem Fall gäbe es z.B. die folgenden Möglichkeiten zu verfahren:

1. Beende den Prozess, der den Page Fault verursacht hat. Diese Methode ist nicht gut, da das Paging für den Benutzer transparent sein soll, d.h. er soll in keiner Art und Weise etwas davon mitbekommen.

2. Swappe einen Prozess aus und reduziere den Grad des Multiprogramming.
3. Ersetze eine Seite, die momentan von keinem Prozess benutzt wird.

Frage 9.12: Wie funktioniert „Page Replacement“?
Wenn es keinen freien Frame mehr gibt, wird beim Page Replacement eine Seite, die momentan von keinem Prozess benutzt wird, durch die selbst benötigte ersetzt. Die Page Fault Serviceroutine muss dabei wie folgt modifiziert werden:

1. Finde die Seite auf der Platte.
2. Finde einen freien Frame:
   a) Gibt es einen freien Frame, so benutze diesen.
   b) Gibt es keinen freien Frame, benutze einen Page Replacement Algorithmus, um ein Opfer (**victim**) zu finden.
   c) Schreibe die Seite des Opfer auf die Platte und ändere entsprechend einige Tabellen (Seite ungültig und wo sich diese jetzt findet).
3. Lese die Seite in den jetzt freien Frame und aktualisiere auch hier die entsprechenden Tabellen.
4. Restarte den Prozess.

Frage 9.13: Wie kann das Page Replacement ein wenig beschleunigt werden?
Es kann ein **modify (dirty) bit** benutzt werden, welches angibt, ob eine Seite verändert wurde oder nicht. Dies wird von der Hardware unterstützt. Eine Seite wird dabei nur auf den Sekundärspeicher zurückgeschrieben, wenn sie modifiziert wurde, ansonsten kann sie einfach überschrieben werden. Wobei die entsprechenden Tabellen natürlich trotzdem aktualisiert werden müssen. Dies lässt sich auch bei Seiten, die ausführbaren Code beinhalten anwenden, da diese auch einfach überschrieben werden können.

9.5 Algorithmen zur Seitenersetzung

Frage 9.14: Wie kann die Effizienz eines Page Replacement Algorithmus ermittelt werden?
Die Effizienz eines Page Replacement Algorithmus kann ermittelt werden, indem die Anzahl der Page Fault gezählt wird, die der Algorithmus auf einer bestimmten Abfolge von Speicherreferenzen erzeugt. Im allgemeinen möchte man natürlich einen Algorithmus benutzen, der die geringste Page Fault-Rate besitzt.

Die Speicherreferenzen können dabei entweder mit einem Zufallsgenerator erzeugt werden oder es kann ein reales System beobachtet werden.

Frage 9.15: Was ist ein „Reference String“?
Wenn ein normales System beobachtet wird, werden innerhalb kürzester Zeit Millionen von Speicherreferenzen erzeugt. Diese müssen jedoch nicht alle aufgezeichnet werden, um einen Page Replacement Algorithmus zu evaluieren. Denn es können zwei Beobachtungen gemacht werden:

1. Es müssen nur Seitenzugriffe und nicht komplette Adressen beobachtet werden, da die Größe einer Seite von vornherein bekannt ist.
2. Wenn wir eine Referenz auf Seite p haben, werden sofort folgende Referentzen auf p garaniert keinen Page Fault verursachen.

Ein Beispiel: Die Pagegröße beträgt 100 Bytes pro Seite und die erzeugten Adressen seien die folgenden:

$$0100, 0432, 0101, 0612, 0102, 0103, 0104, 0611$$

Dem entspricht der folgende Reference String:

$$1, 4, 1, 6, 1, 6$$

Frage 9.16: Wie funktioniert „FIFO Page Replacement"?
Beim FIFO Page Replacement wird mit jeder Seite die Zeit assoziert, wann diese in den Speicher geladen wurde. Muss eine Seite ersetzt werden, wird die älteste Seite genommen. Es sei angemerkt, dass die Zeit nicht wirklich mit aufgezeichnet werden muss, da man die Seiten einfach in eine FIFO-Queue packen kann und stets die Seite am Kopf ausgetauscht wird.

FIFO Page Replacement ist einfach zu implementieren und zu verstehen, jedoch kann die Performanz teilweise sehr schlecht sein, da die Seiten einfach stur der Reihe nach ausgetauscht werden, wobei auch häufig benutzte Seite nicht besonders berücksichtigt werden. Weiterhin leidet es an Beladys Anomalie.

Frage 9.17: Was ist Beladys Anomalie?
Unter **Beladys Anomalie** versteht man das Phänomen, dass eine Erhöhung der verfügbaren Frames zu einer Erhöhung der Page Fault Rate führen kann.

Frage 9.18: Wie funktioniert „Optimal Page Replacement"? Warum kann es nicht implementiert werden? Wo gab es schonmal eine ähnliche Situation?
Nach der Entdeckung von Beladys Anomalie haben sich Forscher gefragt, ob es einen optimalen Seitenersetzungsalgorithmus gibt. Ein solcher hat die niedrigste Page Fault-Rate von allen möglichen Algorithmen und leidet auch nicht an Beladys Anomalie. Ein solcher Algorithmus, genannt OPT oder auch MIN, wurde auch tatsächlich gefunden und lautet wie folgt:

> Ersetze die Seite, die am spätesten erst wieder benutzt wird.

Das Problem an diesem Algorithmus ist, dass zukünftiges Wissen über das Aussehen des Reference String benötigt wird. Ein ähnliches Problem gab es schonmal beim SJF Scheduling, wo die niedrigste nächste CPU-Burst-Zeit bekannt sein musste. Aus diesem Grund wird OPT in den meisten Fällen nur als Vergleich für andere Algorithmen herangezogen.

Frage 9.19: Wie funktioniert „LRU Page Replacement"?
LRU steht für **Least Recently Used**. Bei diesem Algorithmus wird die Seite ausgetauscht, die am längsten nicht benutzt wurde. Es wird also die jüngere

Vergangenheit zur Approximation der Zukunft benutzt. Man kann sich diese Strategie auch wie OPT, der in der Zeit rückwärts gerichtet ist, vorstellen. Wie wir noch sehen werden ist eine tatsächliche Implementierung des reinen LRU Algorithmus in Bezug auf den Zeitaufwand zu aufwändig.

Frage 9.20: Welche beiden Verfahren können angewandt werden, um den Zeitpunkt der letzten Benutzung einer Seite herauszufinden?

Zähler: Der CPU wird ein Zähler oder eine logische Uhr hinzugefügt, die bei jeder Speicherreferenz erhöht wird. In der Pagetabelle gibt es zusätzlich ein Feld, in dem gespeichert wird, wann die Seite zuletzt benutzt wurde. Es muss also stets die ganze Tabelle durchlaufen werden, um die Seite mit dem ältesten Eintrag zu finden.

Stack: Die Seitennummern werden in einem Stack gespeichert, der als doppelt verkettete Liste implementiert ist. Wann immer eine Seite benutzt wird, wird der Stack durchlaufen und die Seite von ihrer aktuellen Position aus nach ganz oben auf den Stack gelegt. Auf diese Art und Weise, findet sich die älteste Seite immer ganz unten im Stack.

Frage 9.21: Was ist die Klasse der „Stack-Algorithmen“?

Ein Stack-Algorithmus ist ein Algorithmus, für den gezeigt werden kann, dass die Menge der Seiten, die sich bei n Frames im Speicher befindet, eine Untermenge der Seiten ist, die sich bei $n + 1$ Frames im Speicher befinden würden. Die Klasse der Stack-Algorithmen zeigt niemals Beladys Anomalie.

Frage 9.22: Welche Hardwareunterstützung gibt es für LRU Page Replacement?

Die grundsätzliche Hardwareunterstützung, die gegeben sein kann, ist die Verwendung eines **Reference Bits**. Das Reference Bit einer Seite wird auf 1 gesetzt, wenn die Seite in irgend einer Art und Weise referenziert wird (z.B. durch lesen oder schreiben).

Zu Beginn werden alle Reference Bits vom Betriebssystem auf 0 gesetzt. Nach einer Weile der Ausführung von Prozessen, kann anhand der Reference Bits festgestellt werden, welche Seite im Verlauf benutzt wurde. Obwohl bei diesem Vorgehen der Verlauf der Zugriffe nicht bekannt ist, führt die Verwendung eines Reference Bits zu vielen Algorithmen, die LRU Page Replacement approximieren.

Frage 9.23: Welche Möglichkeiten kennst du, das Reference Bit zur Approximation von LRU Page Replacement zu benutzen?

Im Buch werden drei Möglichkeiten beschrieben:

Additional-Reference-Bits Algorithmus: Hier wird das Reference Bit in regelmäßigen Abständen aufgezeichnet. Dazu kann z.B. ein Byte benutzt werden, bei dem bei jeder Aufzeichnung, das Reference Bit einer Seite an die höchstwertige Stelle geschrieben wird, wobei alle restlichen Bits um eine Stelle nach rechts geshiftet werden, so dass bei jedem Mal das niederwertigste Bit verloren geht. Auf die Art und Weise wird die Geschichte einer Seite aufgezeichnet.

Hat zum Beispiel eine Seite den Wert 10010110 und eine andere Seite den Wert 01011111, so ist die Verwendung ersterer Seite noch nicht so lange her, wie die der zweiten Seite. Werden die Bytes als vorzeichenlose Integer betrachtet, so ist die Seite mit der niedrigsten Nummer die LRU-Seite. Haben mehrere Seiten die gleiche Nummer, werden entweder alle ausgetauscht oder per FIFO-Prinzip vorgegangen. Die Anzahl der Bits kann natürlich variiert werden.

Second-Chance Algorithmus: Der Second-Chance Algorithmus verwendet nur ein Reference Bit. Im Prinzip handelt es sich um einen FIFO Algorithmus. Die Seiten sind in einer zirkulären Queue verwaltet. Soll eine Seite ausgetauscht werden, so wird diese Queue durchlaufen. Wird eine Seite gefunden, deren Reference Bit gleich 0 ist, so wird diese Seite genommen. Eine Seite, deren Reference Bit gleich 1 ist, bekommt eine zweite Chance. Ihr Reference Bit wird auf 0 gesetzt und ihre Ankunftszeit auf die aktuelle Zeit. Sind alle Reference Bits auf 1 gesetzt, so kommen wir irgendwann doch wieder bei der ersten Seite an und sie wird doch ausgetauscht. Es kann jedoch auch vorkommen, dass eine Seite nie ausgetauscht wird, da sie oft genug benutzt wird.

Enhanced-Second-Chance Algorithmus: Das Verfahren des Second-Chance Algorithmus kann noch erweitert werden, indem das *modify bit* ebenfalls berücksichtigt wird. Es ergeben sich dann vier Fälle die auftreten können, wobei das Modify Bit jeweils hinten steht:

1. $(0, 0)$: Weder benutzt noch modifiziert und somit ein optimaler Kandidat.
2. $(0, 1)$: Die Seite wurde in letzter Zeit nicht mehr benutzt, wurde jedoch schon modifiziert. Nicht ganz so gut wie 1, da die Seite erst rausgeschrieben werden muss.
3. $(1, 0)$: Vor kurzem benutzt jedoch nicht modifiziert. Wird bald vielleicht erneut benutzt.
4. $(1, 1)$: Vor kurzem benutzt und modifiziert. Muss rausgeschrieben werden und wird bald vielleicht wieder gebraucht.

Es wird die Seite ersetzt, die sich in der niedrigsten nichtleeren Klasse befindet. Die Queue muss eventuell mehrfach durchlaufen werden.

Frage 9.24: Welche anderen Algorithmen für die Seitenersetzung kennst du?

Desweiteren gibt es zählerbasierte Seitenersetzungsalgorithmen, welche jedoch nicht häufig benutzt werden, sowie **Page-Buffering Algorithmen**.

Die Idee bei den zählerbasierten Algorithmen ist es, jede Referenz auf eine Seite in einem Zähler zu speichern. Dabei gibt es wiederum zwei Möglichkeiten, welche Seite ersetzt wird. Entweder die mit dem niedrigsten Zählerstand oder die mit dem höchsten. Das Problem bei ersterer Vorgehensweise liegt darin, dass eine Seite in der Anfangsphase eines Prozesses eventuell oft benutzt wird,

dann jedoch nie wieder. Dementsprechend ist das Argument für die zweite Vorgehensweise, dass eine Seite mit einem niedrigen Zählerstand vermutlich erst gerade geladen wurde und erst noch benutzt werden muss. Beide Algorithmen approximieren LRU jedoch schlecht.

Die Grundidee bei Page-Buffering Algorithmen ist es, einen Pool von freien Frames zu halten. Tritt ein Seitenfehler auf, wird wie bisher eine Seite zur Ersetzung (Opfer) gesucht. Die neue Seite wird nun allerdings sofort in einen freien Frame gelesen, so dass der Prozess so früh wie möglich weitermachen kann und nicht erst warten muss, bis die alte Seite rausgeschrieben wurde. Wenn das Opfer später rausgeschrieben wird, wird sein Frame der Liste der freien Frames hinzugefügt.

Eine Erweiterung dieser Idee ist es eine Liste von modifizierten Seiten zu führen. Ist das Paging-Gerät (*paging device*) untätig, werden die modifizierten Seiten auf die Platte geschrieben und ihr Modify Bit resettet.

9.6 Allokation von Frames

Frage 9.25: Warum gibt es eine minimale Anzahl von Frames. die jeder Prozess mindestens benötigt?

In jedem Betriebssystem gibt es eine minimale Anzahl von Frames, die jeder Prozess zur korrekten Ausführung benötigt und diese Anzahl ist durch die verwendete Computer-Architektur vorgegeben. Es ist die maximale Anzahl an Seiten die für einen Befehl benötigt werden können. Auf einer Maschine, bei der jeder Speicherzugriff aus höchstens einer Speicherreferenz besteht, wäre die minimale Anzahl an Frames zum Beispiel zwei: einen für die Instruktion und einen für die Speicherreferenz. Werden indirekte Speicherzugriffe erlaubt erhöht sich die Anzahl auf drei.

Weiterhin gibt es bei einigen Architekturen Befehle, die auf mehr als einem Wort arbeiten. Daher kann es vorkommen, dass eine solche Instruktion die Seitengrenze überschreitet.

Das Worst-Case Szenario besteht auf einer Architektur, welche mehrere Ebenen der Indirektion zulässt. Dies kann so realisiert werden, dass in jedem 16-Bit-Wort 15 Bit für die Adresse stehen und ein Bit, ob der Zugriff direkt oder indirekt ist. Auf einer solchen Architektur müssen die Ebenen der Indirektion begrenzt werden, was zum Beispiel durch einen Zähler realisiert werden kann, der zum Betriebssystem trappt, wenn er den Wert 0 erreicht.

Frage 9.26: Welche Methoden der Seitenallokation kennst du?

Grob kann zwischen gleichgewichteter Allokation (**equal allocation**) und proportionaler Allokation (**proportional allocation**) unterschieden werden:

Gleichgewichtet: Bei dieser Art der Allokation werden die Frames gleichmäßig auf die vorhandenen Prozesse aufgeteilt. Gibt es m Frames und n Prozesse, so bekommt jeder Prozess $\lfloor \frac{m}{n} \rfloor$ Frames. Verbleibende Frames können z.B. einen Pool an freien Frames bilden.

Proportional: Prozesse haben unterschiedliche Speicheransprüche. Bei der proportionalen Allokation wird versucht, dem Rechnung zu tragen. Ist s_i die

Größe des virtuellen Speichers, den p_i benötigt, dann ist der Gesamtbedarf an virtuellem Speicher S gegeben durch:

$$S = \sum s_i$$

Bei der proportionalen Allokation bekommt dann jeder Prozess a_i Frames, wobei m die Gesamtanzahl an verfügbaren Frames ist:

$$a_i = \frac{s_i}{S} \cdot m$$

Frage 9.27: Was ist der Unterschied zwischen globaler und lokaler Seitenersetzung?

Globale Seitenersetzung erlaubt es einem Prozess, eine Seite aus der Menge aller Seiten zu ersetzen. Dies bedeutet, dass die zu ersetzende Seite auch einem anderen Prozess gehören kann. Lokale Ersetzung erlaubt es einem Prozess nur, eine Seite aus der Menge seiner eigenen Seiten zu ersetzen.

Der Nachteil der globalen Seitenersetzung ist, dass ein Prozess nicht mehr seine eigene Page Fault-Rate kontrollieren kann, da diese auch durch andere Prozesse bestimmt wird. Andererseits kann es sein, dass ein Prozess garnicht so viele Seiten braucht, so dass diese bei der globalen Ersetzung von einem anderen Prozess benutzt werden können. Daher wird globale Ersetzung der lokalen vorgezogen.

9.7 Trashing

Frage 9.28: Was versteht man unter „Thrashing“?

Wenn die Anzahl der Frames eines Prozesses unter das Minimum fällt, so muss der Prozess unterbrochen werden und kann erst weiter ausgeführt werden, wenn wieder genügend Seiten zur Verfügung stehen. Es kann jedoch auch passieren, dass die Ausführung eines Prozesses zwar technisch möglich ist, der Prozess jedoch so wenig Seiten hat, dass er mehr Zeit mit dem Paging als mit der Ausführung selbst verbringt. Dies ist dann der Fall, wenn eine Seite ersetzt werden muss, die jedoch kurze Zeit später wieder benötigt wird. Diese hohe Aktivität des Paging wird auch **Thrashing** genannt.

Frage 9.29: Wie kann es zu „Thrashing“ kommen?

Ein mögliches Szenario ist das folgende, welches in frühen Systemen, die Paging benutzt haben beobachtet wurde: Das Betriebssystem überwacht die CPU-Nutzung. Ist diese zu niedrig, wird der Grad des Multiprogramming erhöht und neue Prozesse in das System gebracht. Weiterhin wird ein globales Seitenersetzungsverfahren benutzt. Nun tritt einer der Prozesse z.B. in eine neue Phase seiner Ausführung ein, in welcher er mehr Frames braucht, als er schon hat. Diese Frames nimmt er nun anderen Prozessen weg, welche diese jedoch selber brauchen. Diese Prozesse lösen nun auch Page Faults aus und nehmen sich gegenseitig die Seiten weg. Dabei wird der Sekundärspeicher in Anspruch genommen und nach und nach finden sich fast alle Prozesse in dessen Queue, so dass kaum noch Prozesse ausgeführt werden und die CPU-Nutzung sinkt.

Das Betriebssystem wiederum erhöht den Grad des Multiprogramming und da die neuen Prozesse ebenfalls Frames benötigen erhöht sich die Zahl der Page Faults noch mehr, so dass bald wieder alle Prozesse in der Queue sind und die CPU-Nutzung gegen Null geht.

Frage 9.30: Was kann gegen Thrashing unternommen werden?

Eine Möglichkeit besteht darin nur lokale Ersetzungsstrategien zu benutzen. Ein Prozess der thrasht wird dann nicht auch noch andere Prozesse zum thrashen bringen. Allerdings wird auch die Page Fault Service Zeit für andere Prozesse beeinflusst, da ein thrashender Prozess sich den größten Teil seiner Zeit in der Queue für das Paging-Gerät befindet.

Sinnvoller wäre es also herauszufinden, wie viele Frames ein Prozess tatsächlich braucht. Dazu kann z.B. die Working Set-Strategie verwendet werden, welche vom Lokalitätsmodell (**locality model**) ausgeht.

Frage 9.31: Was ist das Lokalitätsmodell (Locality Model)?

Das Lokalitätsmodell besagt, dass sich ein Prozess während seiner Ausführung von Lokalität zu Lokalität bewegt. Eine Lokalität ist eine Menge von Seiten, welche aktiv miteinander benutzt werden („*a set of pages that are actively used together*“). Wenn zum Beispiel eine Subroutine aufgerufen wird, so definiert diese eine neue Lokalität, da Speicherreferenzen in Bezug auf die Instruktionen in dieser Routine, sowie ihren lokalen und einigen globalen Variablen gemacht werden.

Es sei noch angemerkt, dass das Lokalitätsmodell die Grundlage für fast jedes Cachingverfahren (wie z.B. TLBs) ist. Wenn es keine Muster im Zugriff auf Daten geben würde, wäre auch Caching nutzlos.

Ein Prozess thrasht nicht, wenn er genügend Frames für seine Lokalitäten hat. Zu Beginn löst der Prozess Page Faults aus, bis die Lokalität geladen wurde. Dann arbeitet er solange weiter, bis er aus der Lokalität heraus ist.

Frage 9.32: Was ist das Working Set-Modell?

Das **Working Set-Modell** beruht auf der Annahme der Lokalität. Im Modell gibt es einen Parameter Δ, welcher das Working Set-Fenster definiert. Die Idee ist es, die letzten Δ Seitenreferenzen zu untersuchen. Die Menge der Seiten, die sich in den letzten Δ Referenzen befinden, ist das Working Set. Wenn eine Seite benutzt wird, befindet sie sich auch im Working Set. Wird sie nicht mehr benutzt, so wird sie Δ Referenzen nach ihrem letzten Gebrauch aus diesem herausfallen. Somit ist das Working Set eine Approximation der Lokalität eines Prozesses. Die Genauigkeit dieses Modells wird durch Δ bestimmt. Ist Δ zu klein, deckt es nicht die gesamte Lokalität ab. Ist es zu groß, kann es mehrere Lokalitäten überlappen. Ein Beispiel ist in Abbildung 33 gegeben. Zum Zeitpunkt t_1 ist das Working Set $WS(t_1) = \{1, 2, 5, 6, 7\}$, während es zum Zeitpunkt t_2 nur noch aus zwei Seiten besteht, nämlich $WS(t_2) = \{3, 4\}$.

Die wichtigste Eigenschaft des Working Set ist seine Größe. Wenn WSS_i die Größe des Working Set von Prozess p_i ist, so errechnet sich der Gesamtbedarf an Frames D als:

$$D = \sum WSS_i$$

Ist der Gesamtbedarf an Frames D größer als die Anzahl der Frames m, so wird

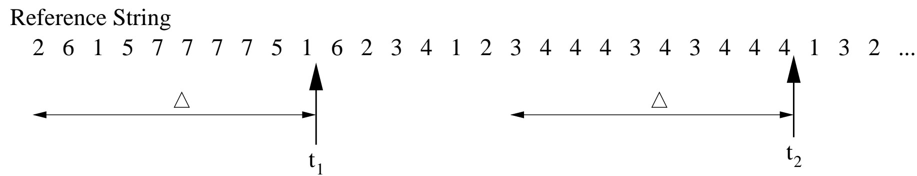


Abbildung 33: Beispiel für das Working Set Modell

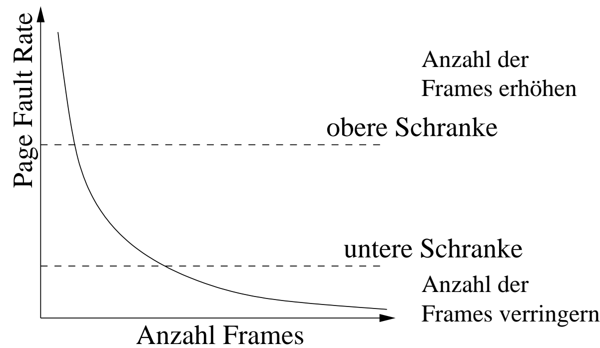


Abbildung 34: Zusammenhang zwischen der Häufigkeit der Page Faults und Thrashing

Thrashing auftreten, da mindestens ein Prozess nicht genug Frames haben wird, sofern die Working Set richtig approximiert wurden.

Die Vorgehensweise des Betriebssystem ist nun wie folgt. Es überwacht die Working Sets aller Prozesse und weist ihnen entsprechend Frames zu. Gibt es ein paar Frames zu viel, kann ein neuer Prozess ins System eingebracht werden. Wird die Gesamtzahl an Frames m überboten, wird ein Prozess unterbrochen und zu einem späteren Zeitpunkt wieder fortgesetzt.

Frage 9.33: Wie kann Thrashing über die Häufigkeit von Page Faults kontrolliert werden?

Das Working Set-Modell kann beim Prepaging recht hilfreich sein, ist jedoch nicht so elegant, um Thrashing zu kontrollieren. Ein direkterer Ansatz ist die Berücksichtigung der Häufigkeiten von Page-Faults (**Page-Fault Frequency**).

Ein Prozess, der thrasht, hat eine hohe Page Fault-Rate. Wenn die Page Fault-Rate eines Prozesses sehr hoch ist, braucht dieser anscheinend mehr Frames, ist sie zu niedrig hat er eventuell zu viele Frames. Es wird nun eine obere und eine untere Schranke in Bezug auf die Page Fault-Rate gebildet. Wird die obere Schranke überschritten, bekommt der Prozess einen neuen Frame, wird sie unterschritten, bekommt der Prozess einen Frame abgenommen. Abbildung 34 verdeutlicht die Idee dahinter.

Auch hier kann es passieren, dass ein Prozess unterbrochen wird, um später fortgesetzt zu werden. Nämlich dann, wenn die Page Fault-Rate zu hoch ist, jedoch keine freien Frames mehr verfügbar sind.

9.8 Weitere Faktoren

Frage 9.34: Was versteht man unter Prepaging?

Wird ein Prozess in einem System, welches Pure Demand Paging benutzt, gestartet, so wird er zu Beginn sehr viele Page Faults verursachen, da er die erste Lokalität erst in den Speicher bekommen muss. Dasselbe passiert, wenn der Prozess rausgeswappt wurde und wieder ins System gebracht wird. Die Strategie beim Prepaging besteht nun darin, nach Möglichkeit alle benötigten Seiten auf einmal in den Speicher zu bringen. Wird z.B. ein Working Set Modell benutzt, so kann eine Liste der Seiten im Working Set zu jedem Prozess verwaltet werden, so dass diese nach einem Swap alle auf einmal wieder geladen werden können.

Frage 9.35: Welche Argumente sprechen für große Seitengrößen? Welche für kleine?

Die folgenden Argumente sprechen für große Seiten:

- Der Pagetable wird für große Seitengrößen kleiner.
- Große Seiten minimieren die I/O, da die Latenz- und die Suchzeiten auf der Platte, die eigentliche Transferzeit um einiges überbieten. Daher ist es intelligenter, wenn man schon eine Seite laden muss, gleich eine große Seite zu laden.
- Je größer die Seiten sind, desto kleiner die Anzahl der Pages und damit auch der Page Faults (Extrembeispiel: Seiten der Größe 1 Byte ⇒ Jedes Byte ein Page Fault).

Für kleine Seiten sprechen die folgenden Argumente:

- Der Speicher wird mit kleinen Seiten besser genutzt, da die interne Fragmentierung kleiner ist. Denn im Schnitt wird 50% der letzten Seite nicht genutzt.
- Kleine Seiten spiegeln Lokalitäten besser wieder, was wiederum zu weniger I/O führen kann.

Der Trend geht eindeutig zu größeren Seiten. Ein Grund ist wahrscheinlich, dass die Plattengeschwindigkeit nicht in dem Maße gestiegen ist, wie die Kosten für Hauptspeicher gesunken sind.

Frage 9.36: Warum kann es manchmal notwendig werden, eine Seite im Speicher zu „locken“, d.h. dass eine Seite nicht ersetzt werden darf?

Eine Situation, in der dies notwendig werden kann, ist die, dass I/O mit Hilfe eines I/O-Controllers durchgeführt wird. Ein Prozess macht eine I/O-Anfrage und wird in die Queue des entsprechenden Gerätes gepackt. Die CPU lässt in dieser Zeit einen anderen Prozess ausführen, der Page Faults verursacht und irgendwann die Seiten, die den I/O-Puffer des anderen Prozesses beinhalten, ersetzt. Ist das I/O-Gerät fertig, schreibt es nun die gelesenen Daten in den Frame des anderen Prozesses.

Ein Lösung kann sein, I/O nur in Betriebssystem-Puffer hinein zu erlauben. Dies resultiert jedoch in einem großen Overhead, da ständig zwischen Betriebssystem- und Benutzerspeicher hin- und herkopiert werden muss.

Daher wird oft mit jedem Frame ein Lock-Bit assoziiert, welches angibt, ob ein Frame ersetzt werden darf oder nicht. Im obigen Beispiel würden also erst die Frames, die den Puffer enthalten gelockt, dann die I/O angestoßen und wenn diese fertig ist, die Frames geunlockt.

Desweiteren müssen auch einige Teile des Kernels (oder teilweise auch der ganze Kernel) gelockt werden. So darf zum Beispiel die Seitenersetzungsroutine keine Page Faults verursachen.

Mit dem Frame Lock muss jedoch vorsichtig umgegangen werden, da es sonst passieren kann, dass eine Seite gelockt wird, das Lock jedoch nicht mehr weggenommen wird.

10 File Systems

10.1 Dateikonzept

Frage 10.1: Was ist eine Datei?
Computer benutzen verschiedene Speichermedien, um Informationen zu speichern. Dazu gehören z.B. Disketten, Festplatten, CDs, optische Disks, etc. Eine **Datei (File)** ist eine logische Speichereinheit, die das Betriebssystem benutzt, um von den physikalischen Eigenschaften der Speichergeräte zu abstrahieren. Durch das Betriebssystem wird eine einheitliche Sichtweise auf die Informationsspeicherung gegeben.

Frage 10.2: Welche Eigenschaften können Dateien haben?
Dateien können eine Reihe von verschiedenen Eigenschaften besitzen, wie z.B.:

Name: Damit Dateien unterschieden werden können, besitzt jede Datei einen Namen.

Typ: Dateien können von verschiedenen Typen sein. Dazu zählen z.B. Textdateien, ausführbare Dateien, Quellcodedateien, Musikstreams, etc.

Location: Ein Zeiger auf das Gerät und den Ort, auf/an dem sich die Datei befindet.

Größe: Die Größe der Datei (z.B. in Bytes).

Zugriffsschutz: Information, wer die Datei lesen, schreiben und ausführen darf.

Zeit-, Datums- und Userinformationen: Kann zum Beispiel den Zeitpunkt der Erzeugung, der letzten Änderung und der letzten Benutzung beinhalten.

All diese Informationen selbst sind auch auf dem Sekundärspeicher gespeichert.

Frage 10.3: Welche Operationen sind auf Dateien möglich?
Eine Datei ist ein abstrakter Datentyp. Um Dateien weiter zu definieren, müssen die Operationen, welche durch das Betriebssystem in Form von System Calls auf den Dateien durchgeführt werden können, definiert werden:

Erzeugung: Diese Operation besteht aus zwei Schritten:

1. Platz für die Datei finden.
2. Datei im Dateiverzeichnis (**Directory**) eintragen.

Schreiben: Um in eine Datei zu schreiben, wird ein System Call ausgeführt, der den Dateinamen sowie die zu schreibenden Daten erhält. Das System sucht dann im Directory, wo sich die Datei befindet. Desweiteren besitzt es einen **write pointer**, der jeweils auf die Stelle zeigt, in der geschrieben werden soll. Dieser Zeiger wird nach jedem Schreiben aktualisiert.

Lesen: Diese Operation ist ein System Call, der den Dateinamen sowie die Speicherstelle, an die der Dateiinhalt hingeschrieben werden soll, erhält. Auch hier wird erst das Verzeichnis nach der Datei durchsucht. Anschließend kann diese über einen **read pointer** gelesen werden. Da Dateien meist entweder gelesen oder geschrieben werden, benutzen viele Systeme nur einen Pointer, der an die aktuelle Position in der Datei zeigt und über den gelesen und geschrieben werden kann.

Positionierung in einer Datei: Diese Operation wird auch **File Seek** genannt und setzt den entsprechenden Pointer an eine Stelle in der Datei.

Löschen: Ein System Call, bei dem der von der Datei allokierte Speicher wieder frei gegeben wird und der Eintrag im Verzeichnis gelöscht wird.

Verkürzung: Oft wollen Benutzer die Eigenschaften einer Datei beibehalten und nur ihren Inhalt komplett löschen. Dies kann über diese Operation gemacht werden.

Frage 10.4: Zeige, wie mit den grundlegenden Operationen eine Datei kopiert werden kann!

Die zu kopierende Datei wird mit `open` geöffnet. Es wird mit `create` die Zieldatei angelegt. Diese wird ebenfalls mit `open` geöffnet und anschliessend Daten aus der einen gelesen und in die andere geschrieben. Abschließend werden beide geschlossen.

Frage 10.5: Was ist der „Open-File Table"?

Der **Open-File Table** ist eine Tabelle, die vom Betriebssystem benutzt wird, damit es Informationen über eine Datei nicht immer wieder von der Platte einlesen muss, was ein unnötiger Overhead wäre. Wird eine Datei geöffnet, wird also alle ihre relavante Information in einen Eintrag in der Tabelle kopiert. Einige Systeme verlangen, dass der Programmierer explizit einen System Call namens `open` tätigt, um eine Datei zu öffnen, andere kopieren die Information einfach bei der ersten Benutzung einer Datei in die Tabelle.

In einer Mehrbenutzerumgebung werden meist zwei verschiedene Arten von Tabellen benutzt. Jeder Prozess besitzt eine Tabelle seiner geöffneten Dateien (**Per-Process Table**), in der auch die prozessabhängigen Informationen (z.B. Position des Read/Write-Pointers) gespeichert werden. Die Einträge in dieser Tabelle zeigen dann weiter auf eine systemweite Tabelle der geöffneten Dateien (**Systemwide Open-File Table**). In dieser systemweiten Tabelle finden sich alle prozessunabhängigen Informationen zu einer Datei, wie z.B. die Poition auf dem Speichermedium, Zugriffszeiten, Dateigröße, etc. Desweiteren wird ein Zähler verwaltet, welcher angibt, von wievielen Prozessen die Datei geöffnet ist. Wird dieser Zähler 0, so wird der Eintrag der Datei aus der Tabelle gelöscht, damit der Platz von einer anderen Datei verwendet werden kann.

Frage 10.6: Welche Informationen sind mit einer geöffneten Datei verbunden?

File Pointer: Gibt die aktuelle Position in der Datei an, z.B. in Systemen, die keinen Offset als Teil der `read`- und `write`-Operationen bekommen.

File Open Count: Gibt an, von wievielen Prozessen die Datei benutzt wird. Wird bei einem `open` um Eins hochgezählt und bei einem `close` um Eins erniedrigt.

Disk Location: Gibt an, wo sich die Datei auf dem Speichermedium befindet.

Frage 10.7: Wie kann „Memory Mapping" im Zusammenhang mit Dateien benutzt werden? Was sind die Vorteile?

Einige Systeme erlauben es, den Inhalt einer Datei in den virtuellen Speicher abzubilden, d.h. ein Teil des logischen Adressraums wird mit einer Datei assoziiert. Die Datei wird dann einfach über Manipulation des Speichers manipuliert. Auf diese Art und Weise ist es auch möglich, Dateien zwischen verschiedenen Benutzer zu teilen. Das Teilen funktioniert dabei wie beim Paging, indem die verschiedenen logischen Adressen der verschiedenen Benutzer (oder Prozesse) auf dieselben Pages zeigen. Dadurch werden Veränderungen sofort bei allen sichtbar. Dieses Sharing ist ebenfalls ein gutes Beispiel für die Verwendung von Mutual Exclusion-Techniken, da der Zugriff auf den Speicher koordiniert vollzogen werden muss.

Frage 10.8: Welche Möglichkeiten gibt es den Dateitypen festzustellen?

In einigen Betriebssystemen wird der Dateityp im Namen festgehalten. In MS-DOS gibt es z.B. das 8.3-Format, wobei die letzten 3 Zeichen den Dateitypen angeben (z.B. `.exe`,`.com`,`.bat`, etc.). In Windows wird der Dateityp ebenfalls anhand der Endung festgestellt. Weiterhin kann ein Programm (oder mehrere) mit einem Dateityp assoziiert werden, so dass das Programm automatisch gestartet wird, wenn z.B. ein Doppelklick auf die Datei gemacht wurde.

In Unix wird oft eine sogenannte **Magic Number** benutzt, die sich irgendwo in der Datei befindet (meist am Anfang) und welche den Dateitypen angibt.

Frage 10.9: Inwiefern kann es Sinn machen, dass das Betriebssystem Dateitypen erkennt oder wo kann es sogar nötig werden?

Das Erkennen von Dateitypen kann dafür sorgen, dass das Betriebssystem sinnvoll mit Dateien umgehen kann. Es kann dann z.B. eine Warnung ausgeben, wenn ein Benutzer eine binäre Datei drucken will.

Andere Systeme benötigen zumindest einiges grundlegendes Wissen über Dateitypen, wie z.B. das Tops-20-System. Ist in diesem System eine Binärdatei älter als eine ihrer Quelldateien, so wird sie neu kompiliert, damit der Benutzer stets eine aktuelle Version eines Programms benutzt. Das System muss also mindestens Quell- und Binärdateien unterscheiden können.

Frage 10.10: Was spricht dagegen, eine oder mehrere Dateistrukturen durch das Betriebssystem bereitzustellen bzw. vorzugeben?

Zum einen kann ein Betriebssystem durch das Bereitstellen mehrerer Dateistrukturen schnell aufgebläht werden, da Code benötigt wird, um die verschiedenen Dateistrukturen zu unterstützen.

Desweiteren kann es Probleme geben, wenn eine neuartige Applikation, die Daten in einer solchen Art und Weise speichern möchte, wie es vom Betriebssystem nicht vorgegeben ist. Sei zum Beispiel ein System gegeben, welches zwischen ASCII-Dateien und Binärdateien unterscheidet. Wenn ein Benutzer nun

ein Programm geschrieben hat, welches ASCII-Dateien verschlüsselt, so wird die resultierende Datei keine der beiden Strukturen aufweisen. Es wird (höchst wahrscheinlich) keine ASCII-Datei mehr sein, da nicht druckbare Zeichen vorkommen, aber auch nicht ausführbar sein.

UNIX hingegen interpretiert alle Dateien als eine Sequenz von Bytes, wobei diese nicht interpretiert werden. Dies bedeutet maximale Flexibilität, jedoch auch, dass jedes Programm selbst den Code bereitstellen muss, um diese Dateien zu interpretieren.

Das MACINTOSH-System hingegen erwartet, dass eine (ausführbare) Datei aus zwei Teilen besteht: einer **resource fork** und einer **data fork**. In der Data Fork steht der Code des Programms, während in der Resource Fork die von der Anwendung benutzten Resourcen, wie z.B. die Beschriftung eines Buttons, etc. stehen. Da das MAC-OS Tools bereitstellt, diese Resource Fork zu editieren, wird es z.B. sehr einfach Programme zu übersetzen.

10.2 Zugriffsarten

Frage 10.11: Welche zwei Arten des Zugriffs gibt es?
Man kann zwischen sequentiellem und direktem Zugriff unterscheiden:

Sequentiell: Sequentiellen Zugriff kann man sich wie den Zugriff auf ein Band vorstellen. Es gibt eine aktuelle Position an der man sich befindet. Desweiteren kann zum Beginn der Datei oder zum Ende „gespult“ werden.

Direkt: Beim direkten Zugriff kann z.B. direkt auf einzelne Blöcke bzw. logische Einheiten zugegriffen werden. Dabei ist die Blocknummer oft eine relative, d.h. sie wird z.B. in der Form „der 16. Block vom Beginn der Datei“ angegeben. Es sei angemerkt, dass sequentieller Zugriff mit direktem Zugriff simuliert werden kann (die aktuelle Position wird also aktueller Block in einer Variablen gespeichert).

10.3 Verzeichnisstrukturen

Frage 10.12: Wie wird eine Festplatte im allgemeinen organisiert?
Heutige Festplatten sind sehr groß und können einige Gigabytes an Daten halten. Um diese Datenmengen zu verwalten, werden Festplatten in zwei Schritten organisiert. Zuerst wird das Filesystem in verschiedene Partitionen unterteilt. Partitionen kann man sich wie virtuelle Festplatten vorstellen. Es ist möglich eine Festplatte in mehrere Partitionen zu unterteilen. Es ist aber auch möglich, mehrere Festplatten zu einer Partition zusammen zu fassen,

Jede Partition enthält Informationen über die Dateien, die sich in ihr befinden. Diese Dateiinformationen werden in einem Verzeichnis (**directory**) in dieser Partition festgehalten. Ein Verzeichniseintrag beinhaltet z.B. den Namen der Datei, ihre Größe, ihre Position auf der Platte oder auch ihren Typ.

Frage 10.13: Welche Operationen auf einem Verzeichnis sind wünschenswert?

Datei suchen: Wir wollen in der Lage sein, eine bestimmte Datei zu finden.

Datei erzeugen: Es soll möglich sein, neue Dateien zu erstellen und in das Verzeichnis einzutragen.

Datei löschen: Nimmt einen Dateieintrag aus dem Verzeichnis.

Verzeichnis auflisten: Wir wollen uns den Inhalt eines Verzeichnisses und seiner Einträge anschauen können.

Datei umbenennen: Wenn sich der Inhalt einer Datei ändert, kann es sein, dass der neue Inhalt durch einen anderen Namen besser für den Benutzer repräsentiert ist. Daher muss es möglich sein, Dateien umzubenennen. Es kann passieren, dass sich dadurch auch die Position auf der Platte ändert.

Dateisystem traversieren: Es soll möglich sein, das Dateisystem zu durchlaufen, was z.B. beim Anlegen von Backups sinnvoll sein kann.

Frage 10.14: Welche Arten von Verzeichnissen bzw. Verzeichnisstrukturen kennst du?

1. Ein-Ebenen-Verzeichnis (**single-level directory**)
2. Zwei-Ebenen-Verzeichnis (**two-level directory**)
3. Baum-Struktur-Verzeichnis (**tree-structured directory**)
4. Azyklischer Graph (**acyclic-graph directories**)
5. Genereller Graph (**general graph directories**)

Frage 10.15: Wie sieht ein Ein-Ebenen-Verzeichnis aus?
Bei diesem Modell werden alle Dateien in ein Verzeichnis eingetragen. Der Nachteil ist der, dass es nach einer Zeit sehr schwer werden kann, eindeutige Namen zu finden, die auch den Inhalt repräsentieren. Gibt es auf dem System auch noch mehrere Benutzer kann dies sehr unübersichtlich werden.

Frage 10.16: Wie sieht ein Zwei-Ebenen-Verzeichnis aus?
Ein Zwei-Ebenen-Verzeichnis ist im Gegensatz zum Ein-Ebenen-Verzeichnis schon ein gewisser Fortschritt, da hier schon jeder Benutzer ein eigenes Verzeichnis besitzt, in welchem er seine Dateien ablegen kann. Diese Verzeichnis wird **User File Directory (UFD)** genannt und besitzt wiederum einen Eintrag in einem **Master File Directory (MFD)**. Loggt sich ein User ein, wird das MFD nach seinem Loginnamen durchsucht, um sein UFD zu finden.

Frage 10.17: Was ist ein Pfadname? Was ist der Suchpfad?
Das Zwei-Ebenen-Verzeichnis bietet schon einige Vorteile im Vergleich zum Ein-Ebenen-Verzeichnis, jedoch kann es vorkommen, dass man die Dateien eines anderen Benutzers ebenfalls benutzen möchte (z.B. weil man in einem gemeinsamen Projekt arbeitet). Um in diesem Fall an die Datei zu kommen, muss der gesamte Pfad angegeben werden. Pfad deshalb, weil man sich das Verzeichnis

wie einen Baum vorstellt, den man von der Wurzel bis zu einem Blatt (einer Datei) durchläuft. Im Falle eines Zwei-Ebenen-Verzeichnisses würde der Pfad z.B. wie folgt aussehen:

`/username/dateiname`

Damit Systemdateien nicht in jedes Userverzeichnis kopiert werden müssen bzw. umständlich mit kompletter Pfadangabe aufgerufen werden müssen gibt es eine Systemvariable (PATH), in der weitere Pfade angegeben werden können, die durchsucht werden, wenn eine Datei nicht im benutzereigenen Verzeichnis gefunden wird.

Frage 10.18: Was ist ein Baum-Struktur-Verzeichnis?

Eine Baum-Struktur-Verzeichnis ist die Verallgemeinerung eines Zwei-Ebenen-Verzeichnisses. Es wird dem Benutzer erlaubt, selbst neue Verzeichnisse anzulegen, so dass er seine Dateien entsprechend organisieren kann. Ein Unterverzeichnis ist dabei eine spezielle Datei. In jeder Datei wird z.B. über ein Bit angegeben, ob es sich um eine Datei oder um ein Verzeichnis handelt. Verzeichnisse werden mit Hilfe von speziellen System Calls erzeugt und gelöscht.

Desweiteren hat bei diesem System jeder Benutzer ein momentane Verzeichnis (**current directory**). Spezifiziert der Benutzer eine Datei, dann wird zuerst das momentane Verzeichnis durchsucht und dann der Suchpfad. Dieses momentane Verzeichnis kann durch einen System Call geändert werden (den z.B. der Befehl `cd` benutzt).

Frage 10.19: Wann entsteht ein azyklischer Graph?

Ein azyklischer Graph auf der Verzeichnisstruktur entsteht dann, wenn es erlaubt wird, dass sich verschiedene Benutzer Dateien und Verzeichnisse teilen, d.h. wenn diese bei unterschiedlichen Benutzern in evtl. unteschiedlichen Verzeichnissen auftauchen. Um dies zu realisieren, gibt es verschiedene Möglichkeiten, jedoch muss man dabei stets aufpassen, dass die azyklische Natur des Graphen bei all diesen erhalten bleibt.

Es sei noch angemerkt, dass es sich dabei nicht um verschiedene Kopien derselben Dateien bzw. Verzeichnisse handelt, d.h. wenn eine geteilte Datei verändert wird, tritt diese Veränderung an allen Stellen auf, wo diese Datei auftaucht.

Frage 10.20: Welche Möglichkeiten gibt es geteilte Dateien („shared files“) zu realisieren?

Unter UNIX gibt es zum Beispiel die Möglichkeit einen Link (Softlink) auf eine Datei anzulegen. Dies ist ein Verzeichniseintrag, der auf eine andere Datei bzw. auf ein anderes Verzeichnis zeigt. Wird dieser Link referenziert, erkennt das Betriebssystem, dass es sich um einen Link handelt und löst diesen Link auf, um an die tatsächliche Datei zu kommen. Wird die Verzeichnisstruktur traversiert, werden Links ignoriert, um die azyklische Natur zu wahren.

Eine weitere Möglichkeit wäre es, die Einträge in den verschiedenen Verzeichnissen, in denen sich geteilte Daten befinden, zu duplizieren, was jedoch zu Konsistenzproblemen führen kann, wenn eine Datei geändert wird. Desweiteren ergibt sich die Frage, wann Speicherplatz auf der Platte wieder entgültig freigegeben werden kann, wenn eine Datei gelöscht wird. Wird sie einfach gelöscht,

kann es passieren, dass woanders noch ein Verzeichniseintrag auf die Stelle der Datei verweist und später evtl. sogar mitten in eine andere Datei zeigt. Softlinks können entweder gesucht werden, was sehr aufwendig sein kann, oder einfach stehen gelassen werden, bis sie benutzt werden. In diesem Fall wird ihre Benutzung wie ein illegaler Dateiname behandelt, so dass der Benutzer spätestens dann bemerkt, dass die referenzierte Datei nicht mehr existiert. Links selbst können natürlich stets gelöscht werden.

Eine weitere Möglichkeit, das Problem zu lösen, besteht darin, sich für jede Datei zu merken, wieviele andere Dateien sie referenzieren und sie erst dann zu löschen, wenn sie von keiner anderen mehr referenziert wird. UNIX benutzt dieses Verfahren z.B. für sog. **hard links**, indem in jeder Inode die Anzahl der Referenzen gespeichert wird. Desweiteren dürfen Verzeichnisse nicht mehrfach referenziert werden, damit die azyklische Natur des Graphen erhalten bleibt.

Frage 10.21: Wann entstehen generelle Graphstrukturen in einem Verzeichnissystem?

Ein genereller Graph entsteht, wenn es erlaubt wird, mehrere Referenzen auf Verzeichnissen zu haben. Dabei kann es nämlich vorkommen, dass ein Kreis im Graph entsteht und dieser somit nicht mehr azyklisch ist. Es gibt dann jedoch mehrere Probleme, z.B. dass der Referenzzähler eines Verzeichnis oder einer Datei nicht Null ist, es jedoch trotzdem nicht mehr möglich ist noch ein Verzeichnis oder eine Datei zu referenzieren. In diesem Fall muss eine **Garbage Collection** durchgeführt werden, die jedoch sehr zeitaufwendig ist. Dies funktioniert grob wie folgt:

1. Traversieren des Verzeichnisbaums und markieren aller Dateien.
2. Löschen aller nicht markierten Dateien.

10.4 Schutz des Dateisystems

Frage 10.22: Wieso ist Schutz in Dateisystemen notwendig?

In einem Multiuser-System soll nicht unbedingt jeder Benutzer auf alle Dateien eines anderen Benutzers zugreifen können. Daher muss das System einen gewissen Schutz zur Verfügung stellen. Desweiteren kann es z.B. möglich sein, dass ein Benutzer eine Datei schützen lassen möchte, so dass er sie nicht ausversehen löschen kann. Die folgenden Operationen können z.B. geschützt werden:

- Lesen
- Schreiben
- Ausführen
- Ans Dateiende anhängen
- Löschen der Datei
- Auflisten eines Verzeichnisses

Frage 10.23: Wie kann der Schutz des Dateisystems realisiert werden?
Der am weitesten verbreitete Ansatz ist, den Zugriff von der Identität des Benutzers abhängig zu machen. Einer der einfachsten Ansätze ist die Verwendung einer Liste pro Datei, die beinhaltet, wer auf der Datei welche Operationen durchführen darf. Dies ist jedoch sehr unpraktisch, da die Liste sehr lang werden kann. Daher kann es unter Umständen auch sehr anstrengend werden, eine solche Liste überhaupt zu erstellen. Ein weiteres Problem ist, dass ein Verzeichniseintrag damit nun eine variable Länge haben kann, was sich negativ im Platzmanagement niederschlägt.

Daher wird der Zugriff oft in Bezug auf drei Benutzergruppen festgelegt:

Besitzer: Der Benutzer, der die Datei erzeugt hat, ist ihr Besitzer.

Gruppe: Eine Menge von Benutzern, die zu einer Gruppe zusammengefasst sind.

Andere: Alle anderen, die nicht zu den ersten beiden Gruppen gehören.

In Unix kann für jede dieser Gruppen über die `rwx`-Bits festgelegt werden, ob sie Lese-, Schreib- oder Ausführzugriff auf eine Datei hat.

Ein weiterer Ansatz wäre die Benutzung von Passwörtern für Dateien. Dies kann jedoch auch Nachteile haben. Wenn viele Dateien verschiedene Passwörter besitzen, muss man sich diese alle merken. Gibt es für alle Dateien nur ein Passwort, so kann auf alle Dateien zugegriffen werden, wenn es entdeckt wird. Einige Systeme verfolgen daher den Ansatz, dass nur ganze Verzeichnisse mit einem Passwort geschützt werden können.

10.5 Struktur eines Dateisystems

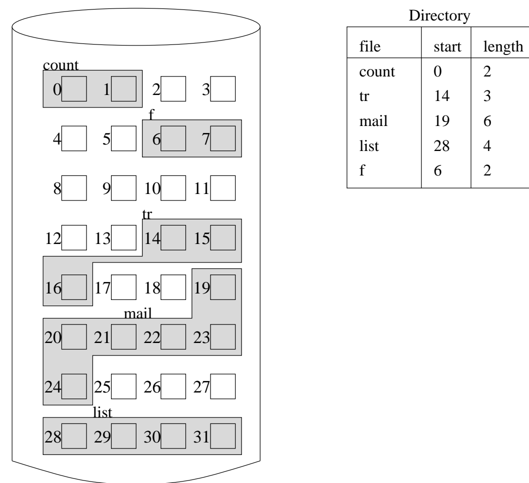
Frage 10.24: Aus welchen „Layern“ besteht ein Dateisystem grob?
Grob kann ein Dateisystem in die folgenden Layer unterteilt werden:

I/O Control: Dies ist die niedrigste Ebene, welche aus den Gerätetreibern und den Interrupt Handlern besteht, welche benutzt werden, um Daten zwischen dem Dateisystem und dem Hauptspeicher zu bewegen.

Der Input besteht hier aus Befehlen wie „Lade Block 123“. Ein solcher Befehl wird dann in Hardwareinstruktionen umgewandelt, mit denen der Hardware Controller des entsprechenden Gerätes angesprochen wird.

Basic File System: Die Aufgabe dieser Komponente ist es, den entsprechenden Gerätetreibern Befehle zum Lesen und Schreiben von Blöcken zu geben. Jeder physikalische Block wird durch seine numerische Diskadresse repräsentiert (z.B. Disk 2, Zylinder 71, Track 3, Sektor 11).

File-Organization Module: Dieses Modul kennt die logischen und die physikalischen Blöcke von Dateien. Dadurch, dass die Art der Allokation auf dem Datenträger bekannt ist, kann aus einem logischen Block ein physikalischer Block errechnet werden. Der Aufwand hierfür kann je nach Art der



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Abbildung 35: Zusammenhängende Allokation

Allokation unterschiedlich sein. Logische Blöcke sind von 0 (oder 1) bis N durchnumeriert.

In diesem Modul wird auch der noch verbleibende freie Platz auf dem Medium gemanagt.

Logical Filesystem: Benutzt die Verzeichnisstruktur, um aus einem symbolischen Namen die Informationen zu bekommen, die es benötigt, um das „File Organization Module“ anzusprechen.

10.6 Allokationsmethoden

Frage 10.25: Welche Arten der Allokation bei Dateisystemen kennst du?

- Zusammenhängend (Contiguous)
- Verkettet (Linked)
- Indiziert (Indexed)

Frage 10.26: Wie funktioniert die zusammenhängende Allokation („Contiguous Allocation“)? Was sind die Vor- und Nachteile?

Bei der zusammenhängenden Allokation werden die Daten in zusammenhängenden Blöcken geschrieben. Ein Verzeichniseintrag besteht hier aus der Adresse des Anfangsblocks und der Anzahl der Blöcke, die die Datei belegt. Ist eine Datei z.B. n Blöcke lang und startet bei Block b, dann belegt sie die Blöcke $b, b+1, b+2, \ldots, b+(n-1)$. Dies ist in Abbildung 35 verdeutlicht.

Der Vorteil bei dieser Art der Allokation ist, dass die Kopfbewegungen minimiert werden, da mehrere Blöcke in einem Track hintereinanderliegen. Es wird nur eine Kopfbewegung benötigt, wenn man sich vom letzten Block eines Tracks zum ersten Block des nächsten Tracks bewegt.

Ein weiterer Vorteil ist der, dass sich dieses Verfahren sowohl für den sequentiellen als auch direkten Zugriff eignet. Beim sequentiellen Zugriff merkt man sich einfach die Plattenadresse des letzten Blocks. Und wenn im Direktzugriff auf den i-ten Block zugegriffen werden soll, findet man diesen einfach an Position $b + i$.

Eines der Hauptprobleme dieses Verfahren ist das Finden von Platz für eine neue Datei. Es handelt sich hierbei um das **Dynamic Storage-Allocation Problem**, welches uns auch schon im Kapitel 8 (Speichermanagement) begegnet ist. Auch hier kann wieder die First-Fit- oder Best-Fit-Strategie benutzt werden, jedoch gibt es auch hier das Problem der externen Fragmentierung. Dies kann zwar ebenfalls durch Compaction gelöst werden, was jedoch bei großen Platten sehr zeitaufwändig werden kann.

Ein weiteres Problem besteht darin, dass die Größe einer neu erzeugten Datei nicht immer von vorherein bekannt ist und einige Dateien im Laufe ihrer Existenz auch noch wachsen (z.B. Log-Dateien). Wenn eine Datei so angewachsen ist, dass sie nun das Loch, in welches sie ursprünglich gepackt wurde, voll ausfüllt, gibt es zwei Möglichkeiten. Das Anwenderprogramm wird mit einer Fehlermeldung beendet und der Benutzer muss es erneut starten und diesmal mehr Platz allokieren. Oder aber das Betriebssystem selbst sucht ein größeres Loch und kopiert die Daten dort hin, wobei der alte Platz wieder freigegeben wird. Davon merkt der Benutzer nichts, außer dass das System unter Unständen langsamer und langsamer wird, wenn dies öfter vorkommt.

Selbst wenn der Platz, den eine Datei benötigen wird, von vornherein bekannt sein sollte, kann es vorkommen, dass der Grad der internen Fragmentierung hoch ist, wenn die Datei nur sehr langsam wächst.

Frage 10.27: Wie funktioniert die verkettete Allokation („Linked Allocation“)?

Die verkettete Allokation löst viele Probleme der zusammenhängenden Allokation. Bei diesem Verfahren besteht eine Datei aus einer verketteten Liste von Blöcken. Ein Verzeichniseintrag besteht aus dem Start- und dem Endblock einer Datei. Soll eine Datei gelesen werden, findet man im Verzeichnis den Startblock. Dieser wird gelesen, wobei an dessen Ende die Adresse des nächsten Blocks zu finden ist, usw. Abbildung 36 zeigt das Prinzip anschaulich.

Die Erzeugung einer neuen Datei führt zu einem Eintrag ins Verzeichnis, wobei jedoch der Startblock auf *null* gesetzt wird. Erst wenn tatsächlich in die Datei geschrieben werden soll, wird ein freier Block gesucht und der Startblock auf dessen Adresse gesetzt.

Wie man schnell sieht, gibt es bei diesem Verfahren nicht das Problem der externen Fragmentierung und desweiteren kann eine Datei beliebig wachsen, da nur ein freier Block gefunden werden muss, der nicht mit anderen Blöcken zusammen hängen muss.

Einer der Nachteile ist der, dass sich dieses Verfahren nur für den sequentiellen Zugriff gut eignet. Will man gezielt einen Block laden, so muss die gesamte Liste durchlaufen werden.

Außerdem wird eventuell viel Platz durch die Zeiger auf die nächsten Blöcke verschwendet. Dies wird durch **Clustering** gelöst, wobei mehrere Blöcke zu

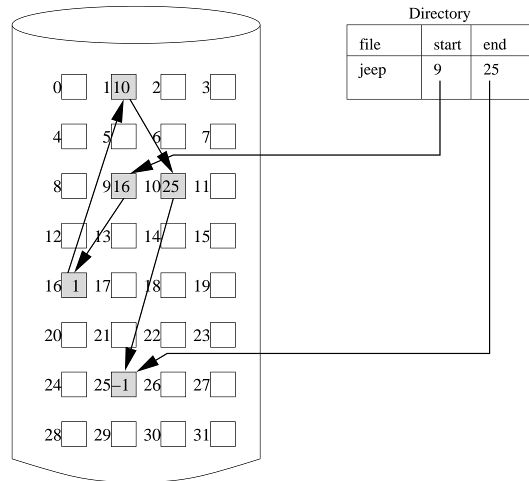


Abbildung 36: Verkettete Allokation

einem Cluster zusammengefasst werden, die dann untereinander verkettet werden. Jedoch steigt damit der Grad der internen Fragmentierung.

Frage 10.28: Wie funktioniert ein „File Allocation Table" (FAT)? Zu welcher Klasse von Allokationsalgorithmen gehört dieses Verfahren?

Das Verfahren des **File Allocation Table**, welches in Abbildung 37 dargestellt ist, gehört ebenfalls zum Verfahren der verketteten Allokation. Nur werden die Zeiger auf die nächsten Blöcke hier nicht in den Blöcken gespeichert. Stattdessen gibt es einen reservierten Platz am Beginn der Partition, wo eine Tabelle liegt, die für jeden Block einen Eintrag besitzt. Ein Verzeichniseintrag für eine Datei zeigt dabei auf den ersten Block dieser Datei. Der nächste Block wird gefunden, indem die Blocknummer dieses Blocks als Index in die FAT benutzt wird. Der Eintrag an dieser Stelle gibt den nächsten Block an. Wird diese Blocknummer als Index benutzt, bekommt man den wiederum nächsten Block, usw. Kommt nach einem Block kein anderer Block mehr, so steht ein spezieller End-Of-File-Wert in der FAT. Nicht benutzte Blöcke haben eine Null als Eintrag in der FAT, Es sei angemerkt, dass diese Verfahren zu vielen zu sehr vielen Lesekopfbewegungen führt, wenn die FAT nicht im Hauptspeicher gecachet wird.

Frage 10.29: Wie funktioniert die indizierte Allokation?

Wird keine FAT benutzt, muss bei der verketteten Allokation stets, die ganze Liste von Blöcken durchsucht werden, um auf einen bestimmten Block zuzugreifen. Bei der indizierten Allokation wird dieses Problem gelöst, indem die Zeiger auf die eigentlichen Datenblöcke alle in einem **Index Block** zusammengefasst werden. Um also auf den i-ten Block zuzugreifen, schaut man in dieser Tabelle nach, welche Adresse an der i-ten Stelle steht.

Ein Nachteil dieses Verfahrens ist, dass unter Umständen sehr viel Platz verschwendet wird, wenn nur ein paar Einträge dieses Index Blocks nicht *null* sind, wie es z.B. bei sehr kleinen Dateien der Fall ist.

Es stellt sich also die Frage, wie groß ein einzelner Indexblock sein soll. Ist

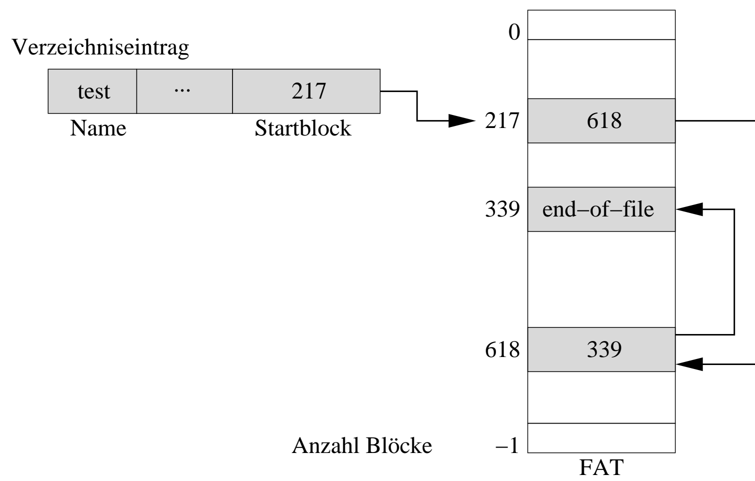


Abbildung 37: Prinzip eines FAT

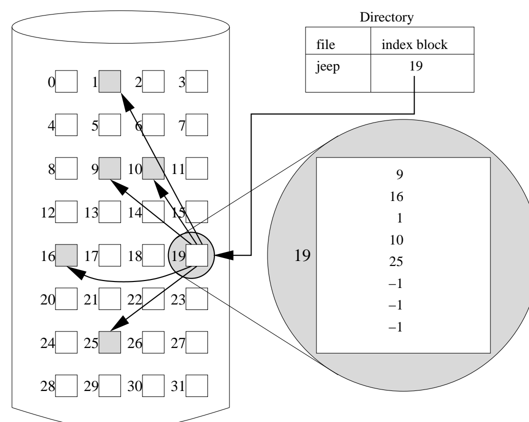


Abbildung 38: Indizierte Allokation

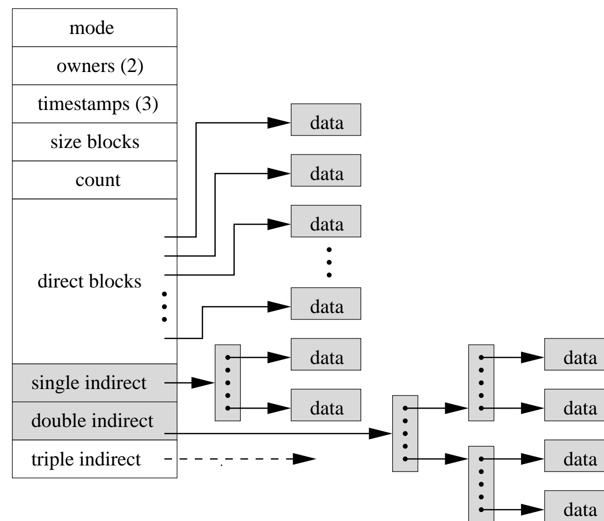


Abbildung 39: Struktur einer Inode

er zu groß, wird zuviel Platz verschwendet. Ist er zu klein, kann es wiederum bei sehr großen Dateien Probleme geben, die gelöst werden müssen. Es bieten sich z.B. die folgenden Lösungen an:

Verkettung: Bei diesem Verfahren werden die Index-Blöcke miteinander verkettet, indem der letzte Eintrag eines Index-Blocks auf die Adresse des nächsten zeigt oder *null* ist wenn es keinen nächsten gibt.

Multilevel Index: Die Idee hier ist ähnlich wie beim Multilevel-Paging. Es gibt einen Index-Block, der auf die Adressen von anderen Indexblocks zeigt, in denen sich die eigentlichen Adressen der Datenblöcke befinden. Auch hier kann man dieses Verfahren wieder in beliebiger Tiefe implementieren.

Kombination: Dieses Verfahren wird z.B. bei BSD UNIX benutzt. Ein Index-Block wird hier Inode genannt. Neben einigen anderen Einträgen enthält eine Inode 12 direkte Blöcke, welche für kleine Dateien ausreichen dürften. Reichen diese nicht gibt es desweiteren noch einen einfach indirekten Block. Reicht dieser nicht aus, so gibt es noch einen zweifach indirekten Block, sowie einen dreifach indirekten Block, für den Fall, dass dieser wiederum nicht reicht.

Es sei angemerkt, dass die Multilevel-Verfahren unter Umständen sehr viel Speicher in Anspruch nehmen können bzw. sehr anwändig werden können, wenn man sich „durch die Platte suchen muss“.

10.7 Verwaltung freier Blöcke

Frage 10.30: Wozu wird die Verwaltung freier Blöcke benötigt? Welche Methoden dazu kennst du?

Wird eine Datei nicht mehr benötigt, so wird sie gelöscht. Im allgemeinen möchte man den Platz der durch diese Datei belegt war, für andere Dateien

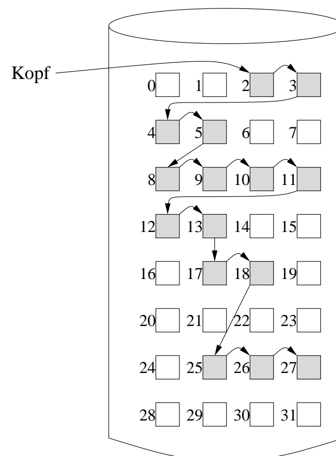


Abbildung 40: Verwaltung von freien Blöcken mit einer verketteten Liste

nutzen können. Daher wird ein Verfahren benötigt, um diese leeren Blöcke zu verwalten. Es gibt unter anderem die folgenden Verfahren:

- Bit Vector
- Verkettete Liste
- Grouping
- Counting

Frage 10.31: Wie funktioniert die Verwaltung freier Blöcke mit einem Bit Vector?

Bei diesem Verfahren, wird jeder Block durch ein Bit in einem Bitvektor repräsentiert. Ist der Block frei, so ist der Wert dieses Bit 1, sonst 0. Der Vorteil bei diesem Verfahren ist, dass viele CPUs effizient Bitoperationen ausführen können, so dass ein freier Block leicht gefunden werden kann. Interpretiert man den Bitvektor als eine Aneinanderreihung von Worten, so sucht man einfach nach dem ersten Wort, welches nicht Null ist. Damit ergibt sich die Nummer des ersten freien Blocks als:

$$\text{Anzahl der Bits pro Wort} \cdot \text{Anzahl der Null-Wörter} + \text{Offset des ersten 1 Bit}$$

Die Bitmaps können bei großen Festplatten sehr groß werden, wobei auch hier Clustering Abhilfe schaffen kann.

Frage 10.32: Wie funktioniert die Verwaltung freier Blöcke mit einer verketteten Liste?

Hier wird ein Zeiger auf den ersten freien Block an einer bestimmten Position auf der Platte gespeichert und im Speicher gecachet. In diesem Block findet sich wiederum ein Zeiger auf den nächsten freien Block usw. Das Travesieren der Liste ist sehr ineffizient, muss jedoch glücklicherweise nicht sehr häufig gemacht werden. Oft genügt es, den ersten freien Block zu bekommen.

Bei der FAT-Methode ist die Verwaltung der freien Blöcke quasi schon „eingebaut“, da man nur die Tabelle durchlaufen muss und den ersten freien Block finden muss.

Frage 10.33: Wie funktioniert die Verwaltung freier Blöcke mit Grouping?
Diese Verfahren hat Ähnlichkeit mit dem Verfahren der indizierten Allokation, nur dass hier ein Block der n Adressen halten kann, $n - 1$ Adressen von freien Blöcken enthält, sowie einen Zeiger auf den nächsten Block dieser Art. Der Vorteil ist, dass man bei diesem Verfahren, sehr schnell mehrere freie Blöcke finden kann.

Frage 10.34: Wie funktioniert die Verwaltung freier Blöcke mit Counting?
Dieses Verfahren macht sich die Tatsache zu Nutze, dass bei einigen Verfahren (wie z.B. der zusammenhängenden Allokation) Plattenplatz oft zusammenhängend frei wird. Es wird sich hierbei die Adresse eines freien Blocks gemerkt, sowie die Anzahl der freien Blocks, die ihm folgen.

10.8 Implementierung von Verzeichnissen

Frage 10.35: Welche Möglichkeiten gibt es Verzeichnisse zu implementieren?
Es werden zwei Möglichkeiten beschrieben:

Lineare Liste: Die einfachste Methode, ein Verzeichnis zu implementieren, ist die Verwendung einer linearen Liste, die die Dateinamen mit Zeiger auf die Datenblöcke enthält. Der Nachteil dieses Verfahrens ist, dass die Suche nach einer Datei sehr ineffizient sein kann, da unter Umständen die gesamte Liste durchlaufen werden muss. Daher wird oft ein Software-Cache benutzt, der die zuletzt benutzen Verzeichnisinformationen enthält. Weiterhin wird die Liste oft sortiert gehalten, so dass mit binärer Suche gesucht werden kann.

Hashing: Damit die Suche nach einer Datei nicht zu lange dauert, kann zusätzlich zu einer linearen Liste eine Hashtabelle benutzt werden, welche die Dateinamen auf Einträge in der linearen Liste abbildet. Nachteile hierbei sind Kollisionen, welche beim Hashing auftreten können, sowie die Tatsache, dass eine Hashtabelle stets eine konstante Größe besitzt. Wird beispielsweise eine Hashtabelle benutzt, die Kollisionen durch lineares Sondieren behandelt, und die Namen auf Zahlen zwischen 0 und 63 abbildet, so muss diese modifiziert werden, wenn der 65. Eintrag gemacht wird. Zum Beispiel, indem sie nun auf die Zahlen zwischen 0 und 127 abbildet.

Eine andere Möglichkeit besteht darin, eine Hashtabelle zu benutzen, die auf lineare Listen abbildet.

10.9 Sonstiges

Frage 10.36: Welche Möglichkeiten gibt es, die Performanz von Platten zu steigern?

- Cache auf dem Disk-Controller, der ganze Tracks cachet.
- Disk-Cache im Hauptspeicher.
- Beim sequentiellen Zugriff Free-behind und Read-ahead.
- RAM Disks.

Frage 10.37: Warum werden Programme benötigt, die die Konsistenz eines Dateisystems prüfen?

Bei einer geöffneten Datei sind die Daten, die sich im Hauptspeicher befinden, oft aktueller als die Daten die sich auf der Platte befinden. Im Falle eines Systemabsturzes kann es also vorkommen, dass sich die Daten in einem inkonsistenten Zustand befinden.

Ein Programm, welches die Konsistenz eines Dateisystems überprüft, vergleicht die Verzeichniseinträge mit den Datenblöcken und versucht alle Inkonsistenzen zu beseitigen. Wie erfolgtreich es dabei ist, kann z.B. vom Allokationsalgorithmus abhängen. Bei der verketteten Allokation kann eine Datei z.B. allein durch ihre Datenblöcke wieder rekonstruiert werden.

Frage 10.38: Warum werden Backups benötigt? Gebe ein geläufiges Schema an!

Fastplatten können wie alle technischen Geräte irgendwann einmal versagen. Da sie oft wichtige Daten enthalten, ist es klug ihren Inhalt von Zeit zu Zeit zu duplizieren, um im Falle eines Versagens weiter darauf zugreifen zu können.

Dazu muss jedoch nicht immer wieder das komplette Datei-System gespeichert werden. Viele Betriebssysteme speichern z.B. den Zeitpunkt der letzten Änderung einer Datei, womit das folgende Schema ermöglicht wird:

- *Tag* 1: Kopierer *alle* Dateien auf das Backup-Medium (**full backup**).
- *Tag* 2: Kopiere alle Dateien, die sich seit Tag 1 verändert haben, auf ein anderes Medium (**incremental backup**).
- *Tag* 2: Kopiere alle Dateien, die sich seit Tag 2 verändert haben, auf ein wieder anderes Medium.
- ...
- *Tag* N: Kopiere alle Dateien, die sich seit Tag $N-1$ verändert haben. Gehe zu Tag 1.

Wenn man wieder zu Tag 1 geht, kann entweder ein altes Medium überschrieben oder ein neues benutzt werden.

11 I/O-Systeme

Frage 11.1: Was versteht man unter dem I/O-Subsystem?
Das I/O-Subsystem ist der Teil des Kernels, der die verschiedenen Ein- und Ausgabegeräte kontrolliert und verwaltet. Es zeichnen sich zwei entgegengesetzte Trends im Bereich der Ein-/Ausgabegeräte ab. Zum einen gibt es einen Trend zur Standardisierung von Geräten, zum anderen werden die Geräte immer verschiedenartiger. Um dieses Problem in den Griff zu bekommen, verwendet der Kernel ein nach außen einheitliches Gerätetreiber-Interface. Die speziellen Eigenarten verschiedener Geräte, werden auf diese Art und Weise nach außen hin versteckt.

Frage 11.2: Welche verschiedenen Möglichkeiten gibt es für die Geräte, um mit dem System zu kommunizieren?
Geräte kommunizieren mit dem System entweder über einen speziellen Verbindungspunkt, der auch **Port** genannt wird, oder über einen Bus. Ein Beispiel für einen Port ist zum Beispiel der serielle Port, über den serielle Daten ausgetauscht werden können. Ein Modem wird beispielsweise am seriellen Port angeschlossen.

Ein Bus ist eine Menge von Leitungen, die von einem oder mehreren Geräten benutzt wird. Dabei wird über ein genau festgelegtes Protokoll festgelegt, wie die Nachrichten, die über den Bus verschickt werden, aussehen müssen. Ein Beispiel für ein Bussystem ist der PCI-Bus, welcher sich in jedem modernen PC findet. Der PCI-Bus verbindet das Prozessor/Hauptspeicher-Subsystem mit den schnellen Ein- und Ausgabegeräten sowie einem Expansion Bus. Der Expansion Bus verbindet die langsameren Geräte, wie z.B. die Tastatur, die Maus und die serielle und parallele Schnittstelle miteinander.

Frage 11.3: Zeichne einen typischen PC-Bus-Aufbau!
Siehe Abbildung 12.1 auf Seite 403.

Frage 11.4: Was ist eine „Daisy Chain“?
Wenn ein Gerät A über ein Kabel mit einem Gerät B verbunden ist, welches wiederum mit einem Gerät C verbunden ist, das über einen Port mit dem Computer verbunden ist, dann nennt man diese Anordnung **Daisy Chain**.

Frage 11.5: Was ist ein Controller?
Ein Controller ist ein elektrisches Gerät, welches einen Port, Bus oder ein anderes Gerät bedienen kann. Ein Controller für den seriellen Port weiß z.B. wie Signale auf die Kabel des seriellen Port gelegt werden können. Ein SCSI-Controller ist ein Controller, der mit einem Bus arbeitet. Ein Beispiel für einen Controller, der ein Gerät bedient, ist ein Festplatten-Controller, welcher weiß, wie die Festplatte angesprochen werden muss, um Daten zu lesen oder zu schreiben. Ein Festplatten-Controller besteht aus einige Chips und Mikrocode und kümmert sich auch um Aufgaben wie das Mapping beschädigter Sektoren, Prefetching, Pufferung und Caching.

Frage 11.6: Welche Möglichkeiten gibt es für den Prozessor einen Controller anzusprechen?

Jeder Controller besitzt Register, in welche Daten und Kontrollsignale geschrieben werden können. Der Prozessor kommuniziert mit dem Controller, indem er diese Register schreibt und liest. Es gibt es zwei Möglichkeiten diese Register zu schreiben:

Spezielle I/O-Befehle: Eine Möglichkeit Daten in die Register zu bewegen, ist die Verwendung spezieller I/O-Befehle, welche ein Byte oder ein Wort zu einer I/O-Portadresse transferieren. Eine solche Portadresse wiederum steht für ein Controllerregister.

Memory-mapped I/O: Bei dieser Methode werden die Register des Controllers in den Adressraum des Hauptspeichers abgebildet, so dass diese Register mit normalen Speicherbefehlen manipuliert werden können.

Frage 11.7: Welche Register besitzt ein I/O-Port im Normalfall mindestens?

Ein I/O-Port besitzt im Normalfall mindestens die folgenden vier Register: `status`, `control`, `data-in` und `data-out`. Das Statusregister kann von Host gelesen werden und beinhaltet Informationen über den momentanen Zustand des Controllers (z.B. ob es einen Fehler gab, ob ein Byte gelesen werden kann, etc.). Das Kontrollregister wird benutzt, um den Controller zu steuern oder um ein Gerät in einen anderen Zustand zu versetzen. Über das `data-out`-Register werden Daten geschrieben und über das `data-in`-Register gelesen.

Einige Controller besitzen FIFO-Chips, so dass mehrere Bytes an Input und Output gehalten werden können.

Frage 11.8: Wie funktioniert „Polling“?

Um Polling zu erklären, wird vom folgenden vereinfachenden **Handshaking**-Protokoll ausgegangen. Der Host und der Controller benutzen zwei Bits, um ihr Consumer/Producer-Verhältnis zu regeln. Ein `busy`-Bit, welches benutzt wird, um anzuzeigen, ob der Controller beschäftigt ist und ein `command-ready`-Bit, welches vom Host benutzt wird, um anzuzeigen, dass ein Befehl ausgeführt werden kann. Das Protokoll benutzt die folgenden sechs Schritte:

1. Der Host liest das `busy`-Bit, bis es nicht mehr gesetzt ist.
2. Der Host setzt das `write`-Bit im `command`-Register und schreibt ein Byte ins `data-out`-Register.
3. Der Host setzt das `command-ready`-Bit.
4. Wenn der Controller das `command-ready`-Bit bemerkt, setzt er das `busy`-Bit.
5. Der Controller liest das `command`-Register, um zu sehen, was er machen soll. Er liest in diesem Fall den `write`-Befehl, liest das `data-out`-Register und vollzieht die I/O.
6. Im Fall, dass kein Fehler aufgetreten ist, wird das `error`-Bit auf Null gesetzt und ebenso das `command-ready`-Bit. Abschließend wird noch das

busy-Bit auf Null gesetzt, so dass der Host evtl. den nächsten Befehl schreiben kann.

Im ersten Schritt dieses Protokoll passiert das eigentliche Polling. Ein status-Bit wird immer und immer wieder gelesen, bis es Null wird. Wenn der Controller und das Gerät schnell sind, kann diese Methode gut benutzt werden. Sind diese jedoch relativ langsam, kann es vorteilhafter sein, eine andere Aufgabe zu erledigen. Dann kann es jedoch passieren, dass ein Puffer überläuft, wenn die CPU sich nicht wieder schnell genug mit dem Gerät bzw. Controller befasst. Aus diesem Grund werden Interrupte benutzt.

Frage 11.9: Welche Instruktionen reichen in der Regel, um ein Gerät zu pollen?

In der Regel reichen drei Instruktionen, um ein Gerät zu pollen: `read` auf einem Geräteregister, logisches `and`, um ein Status-Bit zu extrahieren und ein `branch`, wenn das Ergebnis nicht Null ist.

Frage 11.10: Was ist ein Interrupt und warum werden sie benutzt?

Das Grundprinzip eines Interrupts ist, dass sich Geräte bei der CPU melden, wenn diese auf ein Ereignis reagieren soll (z.B. erfolgreiche Beendigung einer Ein-/Ausgabe). Der grundlegende Mechanismus funktioniert wie folgt:

Die CPU-Hardware besitzt ein Kabel, welches **Interrupt Request Line** genannt wird, das nach jeder ausgeführten Instruktion getestet wird. Stellt die CPU fest, dass ein Controller einen Interrupt ausgelöst hat, speichert sie ihren Zustand ab und führt den **Interrupt Handler** aus. Dieser erkennt, welches Gerät den Interrupt ausgelöst hat, führt die nötigen Aktionen aus und sorgt mit einer `return from interrupt`-Instruktion dafür, dass die CPU genau an der Stelle weitermacht, an der sie unterbrochen wurde.

Frage 11.11: Wozu können Interrupte, abgesehen von der Benachrichtigung der CPU durch Controller, noch genutzt werden?

Interrupte können desweiteren für die folgenden drei Aufgaben benutzt werden:

- Anzeigen von Page Faults
- System Calls
- Kontrollfluss im Kernel steuern

Frage 11.12: Welche Eigenschaften sollte moderne Interrupt-Hardware mindestens bieten und warum?

Moderne Interrupt-Hardware sollte mindestens drei Eigenschaften besitzen:

1. Interrupte sollten verzögert behandelt werden können, z.B. wenn sich die CPU gerade in einer Critical Section befindet.
2. Der entsprechende Interrupt Handler sollte schnell und effizient gefunden werden können, ohne dass alle Geräte gepollt werden müssen.
3. Es sollte zwischen höher- und niederwertigen Interrupte unterschieden werden können (**multilevel interrupts**).

Frage 11.13: Was ist der Unterschiede zwischen maskierbaren und nicht-maskierbaren Interrupten?

Maskierbare Interrupte sind Interrupte, die von der CPU ausgestellt werden können. Über diese Interrupte melden sich die Geräte-Controller bei der CPU. Nicht-maskierbare Interrupte sind Interrupte, die nicht ausgestellt werden können und die in der Regel schwere Fehler anzeigen (Division durch Null, etc.).

Die Intel-Architektur verfügt z.B. über 32 nicht-maskierbare Interrupte und 224 maskierbaren.

Frage 11.14: Was ist der Interruptvektor?

Der Interruptvektor ist ein Array von Adressen, welches sich in der Regel im unteren Speicherbereich befindet. In diesem Vektor befinden sich die Adressen von verschiedenen Interrupt Handler-Routinen. Der Interrupt-Mechanismus erlaubt die Angabe von Adressen, die als Index in dieses Array benutzt werden, so dass ein Gerät bzw. Controller sofort seinen entsprechenden Interrupt Handler aufrufen kann, ohne dass erst alle Geräte gepollt werden müssen.

Da Computer jedoch in der Regel über mehr Geräte verfügen als Interruptvektor-Einträge vorhanden sind, benutzt man das Verfahren des **Interrupt Chaining**, bei dem jedes Element im Interruptvektor auf den Kopf einer Liste von Interrupt Handlern zeigt. Dieses Verfahren ist ein guter Kompromiss zwischen einem Interrupt Handler und einem riesigen Interruptvektor.

Frage 11.15: Nenne einige Beispiel für Interrupte!

- Divide Error
- Invalid Opcode
- Page Fault
- Floating Point Error

Frage 11.16: Wozu sind verschiedene Interrupt-Prioritäten gut?

Verschiedene Interrupt-Prioritäten erlauben es der CPU niedrig-priore Interrupte zugunsten von Interrupten hoher Priorität zu verschieben, ohne dass alle Interrupte maskiert werden müssen. Desweiteren kann bei diesem Mechanismus ein Interrupt mit einer höheren Priorität eine Interrupt mit niedrigerer Priorität preempten.

Frage 11.17: Wie wird ein System Call mit Hilfe von Interrupten realisiert?

Soll ein System Call ausgeführt werden, baut die System Call Routine eine Datenstruktur, um dem Kernel die für den System Call benötigten Daten zu übergeben und führt dann eine spezielle Instruktion aus. Diese Instruktion ist ein **Software Interrupt**, welcher auch **Trap** genannt wird. Zusätzlich besitzt diese Instruktion noch einen Operanden, über welchen der gewünschte Kerneldienst spezifiziert werden kann.

Der Software-Interrupt wird wie ein „normaler“ Interrupt an einen Handler dispatcht. Auch hier wird der Zustand der CPU gespeichert, die Aufgabe vom

Handler erledigt und anschliessend ein `return from interrupt` ausgeführt. Allerdings besitzen Traps niedrigere Prioritäten, da es unter Umständen wichtiger sein kann, erst ein Gerät zu bedienen, bevor sein Puffer überfließt.

Frage 11.18: Wie können Interrupte genutzt werden um den Kontrollfluß im Kernel zu regulieren?

Wenn z.B. Daten von einer Festplatte gelesen wurden, müssen diese noch vom Kernelspeicher in den Benutzerspeicher kopiert werden. Es kann nun jedoch sein, dass schon wieder ein neuer Leseauftrag für das Gerät vorliegt, so dass es effizienter wäre, erst diesen zu starten, da das Lesen von einer Festplatte um einiges länger dauert als das kopieren im Hauptspeicher. Aus diesem Grund wird die Komplettierung der Leseoperation einer Festplatte oft als ein Paar von Interrupten realisiert.

Ein Interrupt hoher Priorität kümmert sich darum, ob der Transfer geklappt hat, startet eventuell den nächsten Transfer und löst einen Interrupt niedriger Priorität aus. Dieser niederpriore Interrupt für an einem unter Umständen späteren Zeitpunkt die Kopierarbeit aus. Die Idee hier ist also, dass die Hardware sich in diesem Fall darum kümmert, ob sofort kopiert werden kann oder ob erst wichtigere Interrupte bedient werden.

Frage 11.19: Was versteht man unter DMA?

Bei Festplatten, Netzwerkkarten und anderen Geräten, die ein große Anzahl von Daten bewegen müssen, wäre es verschwenderisch, die CPU damit zu belasten nach und nach jedes Byte aus dem Controller auszulesen. Ein solches Vorgehen wird **Programmed I/O (PIO)** genannt.

Stattdessen wird für diese Arbeit ein spezieller Controller, ein **DMA Controller** benutzt. In IBM-kompatiblen PCs ist dies der INTEL 8237. Im Groben sieht der Ablauf wie folgt aus: Der Controller, der den Datentransfer durchführen möchte, schickt ein `DRQ`-Signal (`DMA-request`) an den DMA-Controller. Der DMA-Controller fragt dann per `HRQ`-Signal bei der CPU nach dem Bus. Wenn die CPU den Bus nicht mehr benötigt, wird dieser in jedem Fall dem DMA-Controller überlassen. Dies wird dem DMA-Controller per `HLDA`-Signal mitgeteilt, welcher, nachdem er die Zieladresse eingestellt hat, nun ein `DACK`-Signal (`DMA-acknowledge`) an den entsprechenden Controller schickt. Der Controller legt dann das zu transferierende Wort auf den Bus, so dass es im Hauptspeicher gespeichert wird. Der Vorgang wiederholt sich so lange, bis die zu transferierende Anzahl von Bytes transferiert wurde. Die Anzahl der Bytes, sowie die Quell- und Zieladresse wurden von der CPU vorher in einem **DMA Command Block** im Hauptspeicher abgelegt. Nachdem der Transfer beendet ist, nimmt der Controller sein `DRQ`-Signal runter, woraufhin der DMA-Controller das `DACK`-Signal beendet. Desweiteren nimmt der DMA-Controller das `HRQ`-Signal von der Leitung, was die CPU mit einem Herunternehmen des `HLDA`-Signals beantwortet.

Die CPU kann während der gesamten Dauer des Transfers nicht auf den Bus zugreifen, in einigen Systemen läuft sie jedoch so lange weiter, bis ein Zugriff auf den Hauptspeicher stattfindet. Dies funktioniert daher, da die CPU während des Transfers immer noch auf ihre Caches zugreifen kann.

Bei DMA wird entweder direkt auf den Hauptspeicher zugegriffen oder es

wird **Direct Virtual-Memory Access (VDMA)** benutzt, bei dem die Adressen automatisch dem Mapping zwischen virtuellen und physikalischen Adressen unterzogen werden.

Frage 11.20: Wie schafft man es, dass Geräte effizient jedoch auch in einer einheitlichen Art und Weise benutzt werden können?

Dieses Ziel wird erreicht, indem von den genauen Unterschieden zwischen verschiedenen Geräten abstrahiert wird und nur ein paar Gemeinsamkeiten betrachtet werden. Diese Gemeinsamkeiten werden zu einem **Interface** zusammengefasst, welches die Gerätetreiber nach außen hin exportieren und über welches diese benutzt werden können. Die Gerätetreiber selbst kennen jedoch die Eigenheiten des Gerätes, für welches sie geschrieben sind, und können es dementsprechend effizient nutzen.

Es gibt also die folgenden Schichten: Ein Teil des Kernels ist das IO-Subsystem, welches die verschiedenen Arten von Geräten in verschiedene Klassen einteilt (Blockgerät oder Character Stream-Gerät, etc.). Für jedes Gerät gibt es einen Treiber, der die Einzelheiten des Gerätes kennt und ein bestimmtes Interface anbietet. Ein Festplattentreiber wird z.B. das Blockgerät-Interface implementieren. Der Treiber kommuniziert wiederum mit dem Controller des entsprechenden Gerätes, welcher das Gerät selbst nutzt. Eine Abbildung hierzu findet sich im Buch auf Seite 413.

Frage 11.21: Was ist der Vorteil, wenn das I/O-Subsystem unabhängig von konkreter Hardware ist?

Diese Vorgehensweise bietet mehrere Vorteile: Zum einen wird die Aufgabe des Betriebssystem-Entwicklers einfacher und zum anderen bieten sich auch Vorteile für Hardwarehersteller. Denn diese können neue Geräte entweder kompatibel zu existierenden Host-Controller-Interfaces machen (z.B. SCSI) oder einen Gerätetreiber für das Gerät schreiben, welcher eines der Betriebssystem-Interfaces implementiert.

Frage 11.22: In welchen Eigenschaften kann sich Hardware unter anderem unterscheiden?

Verschiedene Hardware kann sich sich in der Art des Transfer-Modus, in der Zugriffsmethode, den Transferzeitpunkten, der Teilbarkeit, der Geschwindigkeit und der Richtung, in der I/O stattfinden kann, unterscheiden. Sie kann daher unter anderem in die folgenden Klassen unterteilt werden:

Character Stream oder Block: Ein **Character Stream-Gerät** transferiert jeweil ein Byte nach dem anderen, während ein **Blockgerät** auf ganzen Blöcken von Bytes arbeitet. So ist ein Terminal z.B. ein Character Stream-Gerät, während eine Festplatte ein Blockgerät ist.

Sequentieller Zugriff oder Random Access: Ein Gerät, welches sequentiell arbeitet, transferiert Daten in einer festen Reihenfolge, während bei einem Random Access-Gerät beliebig in den Daten gesucht werden kann (`seek`). Ein Modem arbeitet sequentiell, während ein CD-ROM ein Random Access-Gerät ist.

Synchron oder asynchron: Ein synchron arbeitendes Gerät transferiert Daten in vorhersagbaren Zeitschritten, während dies bei einem asynchron arbeitenden Gerät nicht der Fall ist. Ein synchrones Gerät ist ein Bandlaufwerk (da das Band in einem konstanten Tempo am Lesekopf vorbeigeführt wird), ein asynchrones Gerät ist die Tastatur.

Teilbar oder unteilbar: Ein teilbares Gerät kann von mehreren Prozessen oder Threads gleichzeitig benutzt werden, während dies bei einem unteilbaren Gerät nicht der Fall ist. Eine Festplatte ist ein teilbares Gerät, ein Drucker nicht.

Geschwindigkeit: Die Geschwindigkeit kann zwischen einigen Bytes pro Sekunde bis hin zu einigen Gigabytes pro Sekunde liegen. Eine Tastatur generiert höchstens ein paar Bytes pro Sekunde, während eine schnelle Netzwerkkarte oder Festplatte Megabytes oder Gigabytes an Daten pro Sekunde generieren können.

Read-Write, Read-Only, Write-Only: Einige Geräte können gelesen und geschrieben werden, während andere entweder nur gelesen oder nur geschrieben werden können. Festplatte sind les- und schreibbar, CD-ROMs nur lesbar und Grafikkarten nur beschreibbar.

Frage 11.23: Wie kann man dafür sorgen, dass auch Geräte benutzt werden können, die nicht in eines der Schemata passen?

Die meisten Betriebssysteme bieten hierfür einen **Escape-** oder **Backdoor-System Call** an. In UNIX und LINUX ist dies der System Call `ioctl` (I/O-Control), welcher drei Parameter entgegen nimmt. Das erste Argument ist ein File Descriptor, der die Applikation mit dem Treiber des entsprechenden Gerätes verbindet, das zweite ist eine Ganzzahl, die die Operation spezifiziert und das dritte ein Zeiger auf eine beliebige Datenstruktur, die genutzt wird um Informationen zwischen dem Treiber und der Applikation auszutauschen.

Frage 11.24: Welche Interfaces zu Geräten bieten Betriebssysteme in der Regel an?

- Block
- Character Stream
- Memory Mapped Files
- Network Sockets
- Timer

Frage 11.25: Was ist das Blockgerät-Interface?

Das Blockgerät-Interface bietet mindestens die Operationen `read`, `write` und, falls es sich um ein Random Access-Gerät handelt, die Operation `seek` an. Normalerweise werden die Gerät über das Dateisystem benutzt, jedoch gibt es noch eine weitere Sichtweise auf Blockgeräte, welche diese als ein Array von Blöcken

auffasst. Diese Art des Zugriffs wird **Raw I/O** genannt und wird teilweise vom Betriebssystem selbst aber auch von Anwendungen wie z.B. Datenbanken benutzt.

Frage 11.26: Wie funktionieren „Memory Mapped Files“?
Bei speicherabgebildeten Dateien wird eine Datei oder ein Teil von ihr in den Hauptspeicher abgebildet. Sie kann dann manipuliert werden, indem in diesen Speicherbereich gelesen oder geschrieben wird. Zur Implementierung von speicherabgebildeten Dateien bieten sich Blockgeräte an, da in diesem Fall derselbe Mechanismus wie beim virtuellen Speicher benutzt werden kann.

Frage 11.27: Was ist das Character Stream-Interface?
Das Character Stream-Interface bietet die Operationen (System Calls) `get` und `put` auf jeweils einem Zeichen an. Auf diesem Interface können Bibliotheken aufsetzen, welche zeilenweise Operationen unterstützen und sich um die Pufferung kümmern.

Typische Beispiele für Geräte, die dieses Interface benutzen, sind Geräte, welche spontan und zeitlich nicht vorhersehbar Daten produzieren. Dazu zählen Tastaturen, Mäuse, Modems und andere Geräte. Auch Ausgabegaräte, wie Soundkarten und Drucker, passen gut in diese Kategorie.

Frage 11.28: Wie funktionieren „Network Sockets“?
Da Netzwerkkarten anders funktionieren als z.B. Festplatten, funktioniert das `read-write-seek`-Interface bei ihnen nicht so gut. Stattdessen bieten die meisten Betriebssysteme, wie z.B. UNIX oder WINDOWS NT das **Socket**-Interface an.

Sockets benutzen die Metapher einer Steckdose für Elektrizität. Jedes Gerät kann in sie eingestöpselt werden und sie so benutzen, um Strom zu beziehen. Analog erlauben es die System Calls des Socket-Interface einer Applikation einen Socket zu erzeugen, in den sich quasi andere Applikationen einstöpseln können, um mit der Applikation, welche den Socket erzeugt hat, zu kommunizieren. Außerdem kann sich eine Applikation mit andere Sockets verbinden und über diese Verbindung Pakete austauschen.

Um Server zu unterstützen, bietet das Socket-Interface den System Call `select` an, welcher eine Menge von Sockets verwaltet und dessen Rückgabe aus Informationen, an welchen Sockets Pakete angekommen sind und welche Pakete versenden können, besteht. Durch diesen Mechanismus entfällt das Polling und das Busy Waiting.

Frage 11.29: Wofür werden Uhren und Timer benutzt?
Uhren und Timer können die folgende Funktionalität bieten:

- Gebe die aktuelle Zeit.
- Gebe die verstrichene Zeit.
- Setze einen Timer, der zum Zeitpunkt T die Operation X ausführt.

Frage 11.30: Was ist ein „Programmable Interval Timer“?

Ein **Programmable Interval Timer** ist ein Stück Hardware, das so eingestellt werden kann, dass nach einer bestimmten Zeitspanne ein Interrupt ausgelöst wird. Dies kann entweder nur einmal geschehen (**one shot**) oder periodisch. Periodische Interrupts werden z.B. beim CPU Scheduling benutzt, damit ein Prozess nicht dauerhaft die CPU in Beschlag nehmen kann. Auch das I/O-Subsystem kann periodische Timer benutzen, z.B. um veränderte Cache-Puffer auf die Platte zu schreiben.

Frage 11.31: Was sind virtuelle Uhren?

Die meisten Betriebssysteme bieten ein Interface für Prozesse an, über welches diese Timer benutzen können. Damit die Anzahl der Timer nicht durch die Hardware begrenzt ist, können virtuelle Uhren benutzt werden. Dabei verwaltet der Kernel (oder der Timer-Gerätetreiber) eine Liste von Interrupten, die von seinen eigenen Routinen und Userprozessen angefordert wurden, welche chronologisch sortiert ist, d.h. die früheste Zeit steht ganz vorn in der Liste. Wenn ein Timer ein Interrupt auslöst, wird die entsprechende Anwendung benachrichtigt, die nächste Zeit aus der Liste eingestellt und der Timer erneut gestartet.

Frage 11.32: Wie genau sind die Uhren und Timer in modernen Systemen?

In modernen Systemen wird die Interruptrate durch das Ticken des Taktgebers generiert und liegt zwischen 18 und 60 Ticks in der Sekunde. Diese Auflösung ist sehr grob, da moderne Computer in dieser Zeit Millionen von Instruktionen ausführen können. Die Genauigkeit ist also durch die Grobauflösung der Uhr, sowie den Overhead, der durch die Verwaltung der virtuellen Uhren entsteht, gegeben.

In den meisten Systemen ist der Taktgeber als ein hochfrequenter Zähler realisiert, dessen Inhalt aus einem Register ausgelesen werden kann. Zwar kann dieser Zähler keinen Interrupt auslösen, er kann jedoch benutzt werden, um die verstrichene Zeit sehr exakt zu messen.

Frage 11.33: Was ist der Unterschied zwischen blockierender und nicht blockierender I/O?

Wenn eine Anwendung einen **blockierenden** System Call tätigt, wird die Ausführung der Anwendung unterbrochen, d.h. sie wird aus der Run Queue genommen und in die Wait Queue gepackt. Wenn der System Call fertig wird, wird sie wieder in die Ready Queue gepackt, so dass sie die Chance bekommt, weiter ausgeführt zu werden. Blockiernde System Calls werden verwendet, da blockierender Code leichter zu verstehen ist als nicht blockierender.

Einige Anwendungen benötigen jedoch nicht blockierende I/O. Zum Beispiel ein Programm, welches Videoframes von der Platte liest und diese gleichzeitig dekomprimiert und anzeigt. Eine Möglichkeit, dies zu implementieren ist die Verwendung von Threads. Während ein Thread die Daten von der Platte einliest (und dabei blockiert), kann der andere diese anzeigen. Diese Technik wurde z.B. von den Solaris-Entwicklern benutzt, um eine Userlevel-Bibliothek für nicht blockierende I/O zu schreiben.

Wieder andere Betriebssysteme bieten nicht blockierende System Calls an. Diese halten die Applikation nicht dauerhaft an, sondern kehren nach einer

gewissen Zeit zurück und geben als Ergebnis z.B. die Anzahl der zur Verfügung stehenden Bytes zurück. Ein gutes Beispiel für einen nicht blockierenden System Call ist der `select`-System Call des Socket-Interface. Eine weitere Alternative sind asynchrone System Calls, welcher jedoch schon komplexer sind.

Frage 11.34: Was ist ein asynchroner System Call?

Ein asynchroner System Call ist ein System Call, der sofort zurückkehrt, ohne auf die Beendigung der I/O zu warten. Der Code der aufrufenden Applikation wird weiter ausgeführt und die Beendigung der I/O der Applikation später, entweder durch setzen einer Variablen oder dem Aufruf einer Callback-Routine, mitgeteilt. Der Unterschied zwischen einem asynchronen und einen nicht blockierenden System Call ist also der, dass der nicht blockierende System Call nach einer kurzen Zeit mit *irgendeinem* Ergebnis zurückkehrt (alle Bytes übertragen, nur ein paar oder auch keines), während bei asynchroner I/O einfach weiter der Code ausgeführt wird und später, nach Vollzug des *kompletten* Transfers, die Applikation benachrichtigt wird.

Frage 11.35: Welche Dienste bietet ein I/O-System im allgemeinen an?

- I/O Scheduling
- Buffering
- Caching
- Spooling
- Gerätereservierung
- Fehlerhandling

Frage 11.36: Was versteht man unter „I/O Scheduling“? Warum wird es benutzt?

Ziel des I/O-Scheduling ist es, die Reihenfolge der I/O-Nachfragen so zu ändern, dass das System effizienter oder auch gerechter benutzt wird. So kann z.B. die Reihenfolge der Plattenlesevorgänge geändert werden, damit der Lesekopf nicht unnötig hin- und herspringt. Welche Möglichkeiten es heir gibt, wird im nächsten Kapitel besprochen.

Desweiteren können auch verschiedene Prioritätsklassen eingeführt werden, so dass z.B. Anfragen des Virtuellen Speicher-Subsystems eine höhere Priorität haben als die Anfragen normaler Benutzerprozesse.

Frage 11.37: Was ist „Buffering“ und aus welchen Gründen wird es benötigt?

Ein Puffer (**Buffer**) ist ein Bereich im Hauptspeicher, welcher benutzt wird, um Daten zwischenzuspeichern, die entweder zwischen zwei Geräten ausgetauscht werden oder einem Gerät und einer Applikation. Es gibt unter anderem folgende Gründe, weshalb Pufferung benötigt wird:

1. Um **Geschwindigkeitsunterschiede** zwischen verschiedenen Geräten auszugleichen.

2. Um **verschiedene Transfergrößen** auszugleichen.
3. Um **Copy Semantics** zu unterstützen.

Frage 11.38: Wie kann Pufferung benutzt werden, um Geschwindigkeitsunterschiede zwischen verschiedenen Geräten auszugleichen?
Als Beispiel nehmen wir an, dass ein Modem Daten, die es empfängt, auf der Festplatte speichern möchte. Da ein Modem in etwa 1000-mal langsamer ist als eine Festplatte, werden die Daten, die es empfängt, erst einmal in einem Puffer zwischengespeichert, damit diese, wenn der Puffer voll ist, in einem Rutsch abgespeichert werden können. Dadurch wird die Festplatte effizienter genutzt. Da das Modem jedoch weiterhin Daten empfangen kann, wenn die „alten“ Daten auf die Platte geschrieben werden, wird ein zweiter Puffer benötigt, in welchen in dieser Zeit, die Daten hineingeschrieben werden können. Ist der zweite Puffer voll, wird wieder der erste benutzt, der zu diesem Zeitpunkt hoffentlich schon auf die Festplatte geschrieben wurde. Diese Technik wird **Double Buffering** genannt.

Frage 11.39: Was versteht man unter „Copy Semantics“ und wie werden Puffer in diesem Zusammenhang genutzt?
Der Begriff **Copy Semantics** wird am folgenden Beispiel erläuert: Angenommen eine Anwendung möchte einen Puffer auf die Festplatte schreiben. Sie ruft den entsprechenden `write`-System Call auf, wobei ein Pointer auf den Puffer sowie die Anzahl der zu schreibenden Bytes mit übergeben wird. Die Frage ist nun, was passiert, wenn der System Call zurückkehrt und die Anwendung die Daten dann ändern. Denn dann kann es passieren, dass die Daten, welche auf die Platte geschrieben werden, nicht mehr die sind, die zum Zeitpunkt des System Calls vorlagen. Mit Copy Semantics wird dieses Problem gelöst, indem die Daten erst in einen Kernelpuffer kopiert und dann von dort aus geschrieben werden.

Frage 11.40: Was versteht man unter Caching?
Ein **Cache** ist ein schneller Speicher, der eine Kopie von Daten enthält, welche selbst auf einem langsameren Medium vorliegen. Der Zugriff auf den Cache ist also schneller als auf das Original. Der Unterschied zu einem Puffer liegt darin, dass die Daten in einem Puffer vielleicht nur einmal in Form des Puffers vorliegen, während bei einem Cache Daten in mehreren Kopien vorliegen können.

Caching wird im Zusammenhang mit I/O-Subsystemen unter anderem verwendet, um Kopien von Blöcken im Hauptspeicher zu halten, so dass bei einem Zugriff auf einen Block nicht unbedingt auf die Festplatte zugegriffen werden muss. Ein solcher Cache ist auch nützlich, wenn ein Block in kurzen Abständen immer wieder geändert wird.

Frage 11.41: Was versteht man unter Spooling?
Ein **Spool** ist ein Puffer, der die Ausgabe eines Programms hält, welche für ein Gerät bestimmt ist, welches keine vermischten Datenströme (z.B. von mehreren Benutzern) verarbeiten kann. Ein Drucker ist ein gutes Beispiel für ein solches

Gerät. Da es vorkommen kann, dass verschiedene Benutzer zur gleichen Zeit etwas drucken wollen, werden die Druckaufträge vom Betriebssystem abgefangen, damit sich diese nicht vermischen können. Die Ausgabe eines Programms wird in einer Datei gespeichert, der Drucker frei wird und der Druckauftrag gedruckt werden kann.

Frage 11.42: Wie werden Fehler vom I/O-Subsystem gehandhabt?
Zuerst einmal gibt es zwei verschiedene Arten von Fehlern: vorübergehende Fehler und permanente Fehler. Ein vorübergehender Fehler ist z.B. ein Netzstau oder ein Lesefehler auf einer Diskette. Oft können diese Fehler durch ein erneutes Probieren behoben werden, z.B. indem erneut versucht wird ein Paket zu versenden oder einen Block zu lesen. Permanente Fehler sind da schon etwas schwerer und führen dazu, dass Teile eines System oder das ganze System nicht mehr funktionieren können. Ein Beispiel für einen solchen Fehler ist z.B. ein defekter Festplatten-Controller.

Generell sollte jeder System Call des I/O-Subsystems Informationen darüber zurück geben, ob eine Aktion erfolgreich durchgeführt werden konnte oder nicht. In Unix ist dies beispielsweise ein Integer namens `errno`.

Frage 11.43: Wie sehen die Kernel-Datenstrukturen des I/O-Subsystems grob aus?

Frage 11.44: Wie wird das Mapping von einem Dateinamen zum entsprechenden Gerät in MS-DOS vollzogen?
Unter MS-Dos kann ein Dateiname aus zwei Teilen bestehen. Der erste Teil ist ein String, der durch einen Doppelpunkt, vom restlichen Dateinamen abgegrenzt ist. Dieser String ist der Name eines Gerätes, auf dem sich die Datei befindet. Er ist fest in das „Betriebssystem“ einprogrammiert. So steht der String `C:` z.B. für die primäre Festplatte.

Durch den Doppelpunkt ist der Namespace des Filesystems komplett vom Namespace der Geräte abgetrennt. Der Dateiname wird benutzt, um im entsprechenden Verzeichnis auf dem Gerät den ersten Eintrag des File Allocation Table (FAT) zu finden.

Frage 11.45: Wie wird das Mapping von einem Dateinamen zum entsprechenden Gerät in Unix vollzogen?
Unter Unix ist das Mapping etwas aufwändiger, da mehrfach gemappt wird. Geräte werden unter Unix im normalen Filesystem-Namespace repräsentiert. Dafür gibt es ein extra Unterverzeichnis namens `/dev`. Da die Geräte daher ebenfalls Attribute haben, kann dadurch der Zugriff auf sie wie der Zugriff auf „normale“ Dateien geregelt werden.

Da aus einem Pfad und Dateinamen also nicht hervorgeht, auf welchem Gerät die Datei liegt, stellt sich also die Frage, woher Unix bzw. später der Treiber „weiß“, auf welchem Gerät sich eine Datei befindet. Dazu gibt es eine sog. Mounttabelle, in der verzeichnet ist, welches Gerät (also z.B. `/dev/hda1`) in welchem Verzeichnis gemountet ist. Es wird in dieser Tabelle nun der längste zum Pfad und Dateinamen passende Präfix gesucht der passt. Auf dem dazugehörigen Gerät befindet sich die Datei. Dieses Gerät wird nun im `/dev`-

Verzeichnis gesucht. Anstatt einer Inode findet sich zu einem Gerät eine **Major-** und **Minor-Nummer**. Die Majornummer gibt an, welcher Treiber für dieses Gerät zuständig ist und die Minornummer, um welches Gerät es sich handelt. Genauer gesagt, ist die Minornummer ein Index in eine Tabelle des Treibers, in der sich die Port- bzw. Memory-Mapping-Adresse befindet.

Frage 11.46: Warum ist I/O ein wichtiger Faktor in Bezug auf System-Performance?

Es gibt verschiedene Faktoren, welche die Effizienz der I/O beeinflussen, z.B.:

- In vielen Fällen werden Prozesse nach einer I/O-Operation geblockt, so dass ein Context Switch nötig wird. Das Scheduling selbst sowie der Context Switch mit allem, was dazu gehört (z.B. das Flushen der CPU-Caches) belasten die CPU stark.
- Der Hauptspeicherbus wird stark belastet, da die Daten oft erst aus dem Controller-Puffer in den Kernelspeicher kopiert werden müssen und anschließend aus dem Kernelspeicher in den Speicher der Applikation.

Frage 11.47: Wie kann die Effizienz von I/O verbessert werden?

- Reduktion der Context Switches. Zum Beispiel durch die Implementierung einiger Userlevel-Dienste (z.B. des Telnet-Daemons oder eines Webservers) im Kernelspace.
- Reduktion der Kopiervorgänge im System (Controller-Puffer zu Kernelspeicher zu Prozessspeicher).
- Reduktion der Anzahl der Interrupte, indem große Transfers gemacht werden oder indem Polling benutzt wird (wenn das Busy Waiting minimiert werden kann).
- Nutzung von DMA-fähigen Geräten.
- Balancierung von CPU, des Speichersubsystems, des Busses und von I/O-Performance, da ein Bottleneck in einem Bereich dazu führt, dass die anderen Bereiche schlecht ausgenutzt werden.

Frage 11.48: Wo kann I/O-Funktionalität überall implementiert werden? Was sind die Vor- und Nachteile?

Es gibt mehrere Schichten, in denen I/O-Funktionalität implementiert werden kann, die alle ihre Vor- und Nachteile haben. Dies sind:

Applikation: Neue I/O-Funktionalität kann mit Hilfe der schon vorhandenen auf dem Applikationslevel erstellt werden. Die Vorteile dieser Vorgehensweise sind erhöhte Flexibilität und dass ein Fehler auf diesem Level nicht das ganze System zum Absturz bringen kann. Desweiteren ist es nicht nötig, ständig neu zu booten bzw. Treiber neu zu laden.

Diese Vorgehensweise hat jedoch auch den Nachteil, dass eine Implementierung auf diesem Level oft sehr ineffizient ist. Dies liegt z.B. daran, dass nicht direkt auf Kernelstrukturen zugegriffen werden kann oder auch an der höheren Anzahl an Context Switches.

Kernel: Wenn sich bei einer Implementierung auf Anwendungsebene gezeigt hat, dass ein Algorithmus es wert ist, effizienter implementiert zu werden, dann kann man ihn auf Kernelebene implementieren. Dies ist jedoch schon um einiges komplizierter als eine Implementierung auf Anwendungsebene, da ein Kernel ein komplexes Stück Software ist. Weiterhin muss eine Kernelimplementierung gewissenhaft debuggt werden, damit keine Systemabstürze oder ähnliches auftreten.

Hardware: Die größte Effizienz wird mit einer spezialisierten Implementierung auf Hardwareebene erreicht. Dabei wird der Algorithmus entweder im Controller oder in der Gerätehardware selbst implementiert. Die Nachteile sind hier, dass Fehler nicht so leicht behoben werden können bzw. Verbesserungen nicht so leicht zu bewerkstelligen sind. Daher muss eine Implementierung von vornherein gut durchdacht sein, was zu einem höheren Entwicklungsaufwand führt.

Zusammenfassend kann also gesagt werden, dass die Effizienz steigt, je mehr man sich der Hardwareebene nähert. Jedoch steigen auch die Entwicklungszeit und -kosten, während die Flexibilität abnimmt.

12 Massenspeicher

Frage 12.1: In welche drei Teile kann ein Dateisystem grob eingeteilt werden?

- Benutzer- und Programmierinterface zum Dateisystem
- Interne Datenstrukturen des Betriebssystem, um das Interface bereit zu stellen
- Hardware, die von Treibern angesprochen wird

Frage 12.2: Warum werden heutzutage fast nur noch Festplatten als Sekundärspeicher benutzt?
Früher wurden auch Bänder als Sekundärspeicher genutzt, jedoch ist die Zugriffszeit auf Bänder viel höher als bei Platten. Daher werden Bänder nur noch als Backup-Medium oder zur Speicherung von riesigen Datenmengen genutzt.

Frage 12.3: Wie ist eine Festplatte aufgebaut?
Eine Festplatte besteht aus vielen magnetisch beschichteten Scheiben **(Platter)**, welche in der Platte übereinander liegend angeordnet sind. Jede dieser Scheibe ist in verschiedene Tracks unterteilt, welche konzentrisch auf der Scheibe angeordnet sind. Jeder Track wiederum besteht aus mehreren Sektoren, welche auf diesem Track hintereinander liegen. Übereinanderliegenede Tracks werden zu Zylindern zusammengefasst.

Frage 12.4: Wie wird eine Festplatte adressiert?
Eine Festplatte wird als ein lineares Array von **logischen Blocks** betrachtet. Ein logischer Block ist dabei die kleinste Übertragungseinheit. Normalerweise beträgt die Größe eines logischen Blocks 512 Bytes, jedoch kann diese Größe bei modernen Platten durch einen **Low-Level Format** geändert werden (z.B. 1024 Bytes).

Frage 12.5: Wie werden die logischen Blocks auf die Sektoren abgebildet?
Sektor 0 ist der erste Sektor des ersten Tracks des äußersten Zylinders. Diese Abbildung wird durch den Track durch fortgesetzt, dann durch die restlichen Tracks des Zylinders und schliesslich von außen nach innen durch die Zylinder.

Theoretisch kann man mit dieser Abbildung aus einem logischen Block eine Plattenadresse, bestehend aus Zylinder, Track und Sektor, berechnen. In der Praxis ist dies jedoch aus zwei Gründen nicht so einfach:

- Platten haben defekte Sektoren.
- Die Nummer der Sektoren pro Track ist nicht konstant **(Zones)**.

Defekte Sektoren werden durch die Hardware automatisch auf Ersatzsektoren umgeleitet.

Frage 12.6: Aus welchen Größen setzt sich die Zugriffszeit zusammen?
Die Zugriffszeit setzt sich aus zwei Größen zusammen: Der **Suchzeit (Seek Time)** und der **Rotational Latency**. Die Suchzeit ist die Zeit, die der

Kopf braucht, um sich zum entsprechenden Zylinder, der den gesuchten Sektor enthält, zu bewegen. Die Rotational Latency ist die Zeit die vergeht, bis der gesuchte Sektor sich tatsächlich unter dem Kopf befindet.

Frage 12.7: Wie berechnet sich die Bandbreite (Bandwidth) einer Festplatte?
Die Bandbreite einer Platte ist die Anzahl der transferierten Bytes dividiert durch die Zeit, die zwischen der ersten Anfrage und dem letzten Transfer vergangen ist.

Frage 12.8: Wie kann die Zugriffszeit und Bandbreite einer Platte verbessert werden?
Die Zugriffszeit und Bandbreite einer Platte kann verbessert werden, indem die Reihenfolge der Bearbeitung der Anfragen optimiert wird.

Steht nur eine Anfrage an, kann diese sofort bearbeitet werden. Ist die Platte bzw. der Controller jedoch gerade beschäftigt, so wird die Anfrage in die Queue der noch ausstehenden Anfragen gestellt. Befinden sich also mehr als zwei Einträge in dieser Queue, so hat man die Wahl, welche Anfrage als nächstes bearbeitet wird.

Frage 12.9: Welche Arten von Disk Scheduling kennst du?

- First-Come-First-Serve Scheduling (FCFS)
- Shortest-Seek-Time-First Scheduling (SSTF)
- SCAN Scheduling
- C-SCAN Scheduling
- LOOK (und C-LOOK)

Frage 12.10: Wie funktioniert FCFS Disk Scheduling?
Beim FCFS Scheduling werden die Anfragen in der Reihenfolge bearbeitet, wie sie eintreffen. Diese Verfahren ist zwar fair, jedoch im allgemeinen nicht das schnellste Verfahren. Der Grund hierfür ist der, dass der Kopf sich unter Umständen ständig von einem Ende der Platte zum anderen bewegt, obwohl es eventuell schlauer wäre erst einmal näher liegende Anfragen zu bearbeiten, wie es z.B. SSTF Scheduling macht.

Frage 12.11: Wie funktioniert SSTF Disk Scheduling?
SSTF basiert auf der Annahme, dass es sinnvoll ist, erst die Anfragen zu bearbeiten, die sich am nähesten zur aktuellen Kopfposition befinden. Nach Beendigung einer Anfrage wählt SSTF also stets den Eintrag aus der Queue, welcher eine minimale Suchzeit hat.

Man sieht recht schnell, dass SSTF dem SJF Scheduling, welches wir aus Kapitel 5 (CPU Scheduling) kennen, recht ähnlich ist. Er hat auch dasselbe Problem wie SJF, nämlich, dass einige Anfragen verhungern können **(Starvation)**. Dies liegt daran, dass Anfragen zu beliebigen Zeitpunkten eintreffen können. Angenommen, es liegen zwei Anfragen vor: Eine bei Zylinder 14 und eine bei Zylinder 186. Während die Anfrage bei Zylinder 14 gerade bearbeitet

wird, kann es passieren, dass eine neue Anfrage in der Nähe dieses Zylinders eintrifft, welche als nächstes bearbeitet wird. Während dieser Bearbeitung kann wieder eine neue Anfrage in der Nähe eintreffen, so dass die Anfrage bei Zylinder 186 wieder nicht bearbeitet wird, usw.

Es gibt jedoch auch einen Unterschied zum SJF. Während das SJF Scheduling beweisbar optimal ist, in Hinblick auf die durchschnittliche Wartezeit, ist das SSTF Scheduling nicht optimal in Hinblick auf die Anzahl der nötigen Kopfbewegungen.

Frage 12.12: Wie funktioniert SCAN Scheduling?

Beim SCAN Scheduling startet der Plattenarm am einen Ende der Platte und bewegt sich hin zum anderen Ende, wobei alle auf dem Weg liegenden Anfragen bedient werden. Am anderen Ende angekommen, „dreht“ der Arm wieder um und bewegt sich wieder zurück, wobei wieder alle auf dem Weg liegenden Anfragen bedient werden. Aufgrund dieser Funktionsweise wird SCAN Scheduling auch **Elevator Algorithm** genannt, da diese Vorgehensweise stark einem Aufzug ähnelt, der zwischen unterstem und obersten Stockwerk pendelt und dabei jeweils nur die Leute mitnimmt, die in dieselbe Richtung wollen, in die der Aufzug fährt.

Eine eintreffende Anfrage, welche kurz vor dem Arm liegt, wird also schnell bedient, während eine Anfrage, welche dahinter eintrifft, etwas länger warte muss.

Frage 12.13: Wie funktioniert C-SCAN Scheduling? Wie ist es motiviert?

Wenn der Kopf beim SCAN Scheduling das eine Ende der Platte erreicht und umkehrt, wird die Dichte der Anfragen in diesem Bereich nicht sehr hoch sein, da dieser Bereich erst vor kurzem bedient wurde. Die höchste Dichte von Anfragen wird wahrscheinlich am anderen Ende der Platte vorliegen, da diese Anfragen am längsten gewartet haben. Um eine einheitlichere Wartezeit zu erreichen wird das C-SCAN Scheduling benutzt.

Auch hier wandert der Lesearm von einem Ende der Platte zum anderen und bedient Anfragen, die auf dem Weg liegen. Kommt er jedoch am Ende an, wird der Kopf sofort wieder an den Anfang bewegt, ohne dass dazwischen liegende Anfragen bedient werden. Der C-SCAN Algorithmus behandelt die Zylinder also wie eine zirkuläre Liste.

Frage 12.14: Was ist LOOK Scheduling?

Bei der Beschreibung des SCAN- und C-SCAN Scheduling wurde davon ausgegangen, dass sich der Arm stets komplett von einem Ende der Platte zum anderen bewegt. In der Praxis werden diese Algorithmen natürlich nicht so implementiert. Gibt es in der Richtung, in der sich der Arm gerade bewegt, keine Anfragen mehr, so kehrt der Arm sofort um, ohne erst bis ans Ende der Platte zu laufen. Diese Versionen des SCAN und C-SCAN Scheduling werden entdprechend LOOK und C-LOOK Scheduling genannt.

Frage 12.15: Warum kann man das Disk Scheduling nicht komplett der Hardware bzw. dem Controller überlassen?

Das Disk Scheduling kann nicht komplett der Hardware überlassen werden, da das Betriebssystem unter Umständen weitere Bedingungen an die Reihenfolge

der Bearbeitung der Anfragen stellen kann. So kann es zum Beispiel sinnvoll sein, dass Demand Paging Vorrang vor Anwendungs-I/O hat oder dass Schreibvorgänge wichtiger sind, wenn der Cache fast voll ist. Um ein System robuster gegen Abstürze zu machen, kann es ebenfalls sinnvoll sein, z.B. auf die Reihenfolge der Schreibvorgänge Einfluss zu nehmen. Es wäre zum Beispiel fatal, wenn erst Daten in einen Block geschrieben werden würden und der Rechner abstürzen würde, noch bevor die Inodes und Liste des freien Speicherplatzes auf die Platte geschrieben werden würde.

Frage 12.16: Was versteht man unter einer „Low Level Formatierung"?
Eine **Low Level** oder auch **physikalische** Formatierung sorgt erst dafür, dass eine Festplatte benutzt werden kann. Dabei wird die Platte in Sektoren aufgeteilt, die der Controller lesen und schreiben kann.

Frage 12.17: Wie sieht die Datenstruktur eines Sektors üblicherweise aus?
Üblicherweise besteht ein Sektor aus einem Header, einem Datenbereich und einem Trailer. Der Header und der Trailer enthalten Informationen, welche vom Controller benötigt werden, wie z.B. die Sektornummer und einen fehlerkorrigierenden Code **(ECC)**. Der Datenbereich enthält die eigentliche Information und ist üblicherweise 512 Bytes groß. Bei einer Low Level Formatierung können jedoch auch andere Größen benutzt werden, wie z.B. 256 oder 1024 Bytes. Bei größeren Sektoren passen zwar weniger Sektoren in einen Track, jedoch werden auch weniger Header und Trailer benötigt, so dass mehr Platz genutzt werden kann.

Frage 12.18: Wie funktionieren fehlerkorrigierende Codes (ECC) in Festplatten?
Wenn der Controller einen Sektor schreibt errechnet er aus dem Datenteil einen Code, welcher in den Trailer geschrieben wird. Wird ein Sektor gelesen, errechnet der Controller erneut den ECC und vergleicht ihn mit dem im Trailer. Sind die beiden nicht identisch, sind die Daten beschädigt worden.

ECCs heißen fehlerkorrigierend, da sie, falls nur ein oder zwei Bits verfälscht sind, deren korrekte Werte errechnen können.

Frage 12.19: Welche zwei Schritte sind nötig, damit ein Betriebssystem eine Platte nutzen kann.
Die folgenden zwei Schritte sind nötig:

Partitionierung: Bei der Partitionierung werden mehrere Zylinder zu einer Partition zusammengefasst, welche wie eine separate Platte behandelt werden können.

Logische Formatierung: In diesem Schritt legt das Betriebssystem seine eigenen Datenstrukturen für das Dateisystem an. Dazu gehören zum Beispiel Listen freien und belegten Platzes (FAT oder Inodes) und ein initiales leeres Verzeichnis.

Frage 12.20: Was versteht man unter „Raw I/O"?
Einige Betriebssysteme geben Programmen die Möglichkeit, eine Partition als ein langes, sequentielles Array von logische Blöcken anzusprechen, ohne ein

Dateisystem dazwischen. Dies ermöglicht es der Anwendung selbst festzulegen, an welcher Stelle Daten abgespeichert werden. Raw I/O umgeht dabei alle Betreibssystem-Dienste, welche für Dateisysteme angeboten werden, wie z.B. den Puffer Cache, Prefetching, Dateinamen, Verzeichnisse, etc.

Frage 12.21: Was ist ein Bootstrap Programm?
Ein Bootstrap Programm initialisiert alle Aspekte eine Systems, wie z.B. die CPU Register, Gerätecontroller und den Inhalt des Hauptspeichers und lädt anschliessend das Betriebssystem. Das Bootstrap Programm ist in den meisten Fällen auf einem **ROM (Read-Only Memory)** gespeichert. Damit bei einer Änderung der Laderoutine nicht das ganze ROM ausgetauscht werden muss, ist auf diesem ROM meist nur ein sog. **Bootstrap Loader**, welcher das restliche Bootstrap Programm von der Festplatte einliest und startet. Soll die Laderoutine geändert werden, muss nur eine neue Version auf die Platte geschrieben werden.

Frage 12.22: Wo findet sich das Bootstrap Programm welches der Bootstrap Loader lädt?
Das restliche Bootstrap Programm findet sich üblicherweise in einer speziellen Partition, den **Bootblöcken (Boot Blocks)**. Der Bootstrap Loader lädt diese Bootblöcke in den Hauptspeicher und führt den Code aus, welcher das restliche Betriebssystem von der Platte liest.

Frage 12.23: Welche (drei) Möglichkeiten gibt es, mit defekten Blöcken umzugehen?

Manuell: Diese Verfahren wird z.B. bei einigen Platten mit IDE Controllern angewandt. Wird bei der Formatierung ein defekter Block gefunden, wird ein spezieller Eintrag in die FAT gemacht, so dass dieser Block nicht mehr genutzt wird. Wird ein Block während der Benutzung defekt, muss ein spezielles Programm gestartet werden, welches nach defekten Blöcken schaut (z.B. `chkdsk`).

Sector Sparing: Sector Sparing (auch **Forwarding**) wird auf moderneren Platten verwendet. Hierbei verwaltet der Controller eine Liste von defekten Blöcken, welche während eines Low Level Formats initialisiert und während der gesamten Lebensdauer aktualisiert wird. Während der Low Level Formatierung werden auch einige spezielle Ersatzsektoren **(Spare Sectors)** angelegt, welche für das Betriebssystem nicht sichtbar sind. Der Controller kann nun angewiesen werden, einen defekten Sektor logisch durch einen Ersatzsektor zu ersetzen.

Da durch dieses Vorgehen Optimierungen durch den Disk-Scheduling Algorithmus zunichte gemacht werden könnten, werden Platten oft so formatiert, dass sich in jedem Zylinder einige Ersatzsektoren befinden, sowie ein Ersatzzylinder.

Sector Slipping: Alternativ zum Sector Sparing kann das Sector Slipping verwandt werden. Ein Beispiel: Der logische Block 17 wird defekt und der erste Ersatzsektor ist der Sektor 203. Dann werden alle Sektoren um einen

Sektor verschoben, so dass Sektor 202 nun auf Sektor 203 abgebildet wird, Sektor 201 auf Sektor 202, und so weiter, bis schliesslich Sektor 17 auf Sektor 18 abgebildet wird.

13 Verteilte Systeme

13.1 Struktur verteilter Systeme

Frage 13.1: Was ist der Unterschied zwischen einem netzwerkfähigen Betriebsystem und einem verteilten Betriebssystem?

Netzwerkfähiges Betriebssystem: Der User ist sich des Netzwerks bewusst. Er kann sich auf andere andere Rechner des Netzwerks einloggen (`telnet`) und Dateien übertragen (`ftp`), jedoch geschieht dies alles nicht transparent.

Verteiltes Betriebssystem: Der Zugriff auf Resourcen der Rechner im Netzwerk geschieht für den Benutzer transparent. Daten und Rechenzeit können auf verschiedene Rechner aufgeteilt werden. Das Betriebssystem kümmert sich dabei um diese Aspekte. Das Verteilen von Daten auf verschiedenen Rechnern wird hier nun Datenmigration genannt und die Verteilung von Rechenzeit Rechenzeitmigration.

Frage 13.2: Was versteht man unter Datenmigration?
Fordert ein Benutzer eine Datei an, die nicht auf dem lokalen Rechner liegt, so hat das Betriebssystem zwei Möglichkeiten, diese Anforderung zu realisieren:

1. Vollständige Datei übertragen
2. Nur benötigte Teile übertragen

Bei Veränderung der Datei muss darauf geachtet werden, dass die Konsistenz erhalten bleibt.

Frage 13.3: Was versteht man unter Rechenzeitmigration?
Es kann manchmal sinnvoller sein, anstatt der benötigten Daten für eine Berechnung die Berechnung selbst auf einen anderen Rechner zu übertragen. Hier kann unterschieden werden, ob nur eine Prozedur auf einem Rechner ausgeführt wird (RPC) oder ganze Prozesse auf einem anderen Rechner gestartet werden. Motivation für ein solches Vorgehen sind:

- Load Balancing
- Beschleunigung der Berechnung
- Nutzung spezialisierter Hardware
- Nutzung spezialisierter Software

Dies kann entweder transparent für den Benutzer geschehen oder der Benutzer muss angeben, in welcher Form die Migration stattfinden soll (z.B. bei der Benutzung spezialisierter Hardware).

Frage 13.4: Was ist ein RPC?
Ein RPC (Remote Procedure Call) ist die Ausführung einer Funktion bzw. Prozedur auf einem entfernten Rechner. Da die Rechner nur über ein Netzwerk

verbunden sind und daher keinen geteilten Speicher besitzen, muss dies über eine nachrichtenbasierte Kommunikation geschehen. Diese funktioniert ähnlich wie die Interprozesskommunikation per Nachrichten. Die Nachrichten werden an einen RPC-Daemon auf dem entfernten System geschickt. In jeder RPC-Nachricht wird ein Port angegeben, sowie der Name der auszuführenden Funktion und eventuell zu übergebende Parameter. Die Funktion wird auf dem entfernten System ausgeführt und eventuelle Ergebnisse an der anfragenden Rechner zurückgeschickt. Damit Funktionen nicht aufgrund duplizierter Pakete mehrfach ausgeführt werden, wird jeder Anfrage ein Zeitstempel hinzugefügt. Die entsprechende Portnummer für einen entfernten Prozeduraufruf kann entweder standardisiert festgelegt werden oder aber es gibt einen sog. Matchmaker-Daemon, d.h. ein Daemon der ein Rendevouz zwischen den beiden involvierten Systemen ermöglicht.

Zur Implementierung bieten sich serverseitig Threads an. Dabei gibt es zwei Möglichkeiten:

- Mehrere Threads warten auf Anfragen und bedienen diese. Ist ein Thread fertig, wird er gelöscht und ein neuer Thread erzeugt, der wiederum auf Anfragen warten.
- Ein Thread wird aufgrund einer ankommenden Anfrage erzeugt und bedient diese.

Frage 13.5: Welche Fehler können in einem verteilten System auftreten und wie können diese erkannt werden?

Es gibt grob drei Arten von Fehlern, die unterschieden werden können:

- Unterbrechung der Verbindung zwischen zwei Rechnern
- Ausfall eines Rechners
- Verlust einer Nachricht

Die beiden ersteren Fehler können über Handshaking-Prozeduren erkannt werden. Dabei schicken sich die Rechner gegenseitig Nachrichten zu, die erkennen lassen, dass alles in Ordnung ist („*I am up*“-Nachrichten). Bemerkt ein Rechner den Ausfall eines anderen Rächers, so werden alle anderen Rechner im System benachrichtigt. Hat der ausgefallene Rechner eine spezielle Aufgabe übernommen, so muss diese eventuell neu verteilt werden. Ist die Verbindung zwischen zwei Rechnern ausgefallen, so müssen beide benachrichtigt werden. Ist eine Site ausgefallen, so muss diese von allen anderen Sites informiert werden, so dass z.B. Informationen oder auch Tabellen aktualisiert werden können.

Frage 13.6: Welche Aspekte spielen beim Design eines verteilten Systems eine Rolle?

Eine der größten Herausforderungen beim Design eines verteilten Systems liegt darin, das System für den Benutzer transparent erscheinen zu lassen. Dazu zählen unter anderem die folgenden Aspekte:

Mobilität des Benutzers: Die Arbeitsumgebung eines Benutzer ist stets dieselbe, egal an welchem Rechner des Systems er arbeitet.

Fehlertoleranz: Der Ausfall einer oder mehrerer Komponenten darf nicht das komplette System zusammenbrechen lassen. Selbst wenn einige Dienste ausfallen, sollte es noch eingeschränkt benutzbar sein.

Skalierbarkeit: Unter Skalierbarkeit eines Systems versteht man die Fähigkeit des Systems, sich an erhöhte Last anzupassen. Unter erhöhter Last sollte die Leistung eines solches Systems langsamer abnehmen als die eines nicht skalierbaren Systems und desweiteren sollten seine Resourcen auch erst zu einem späteren Zeitpunkt ausgelastet sein. Zentrale Dienste, wie z.B. Namensauflösung, sollten vermieden werden. Asymmetrischen Strukturen sollte in skalierbaren Systemen der Vorzug gegeben werden. Im Allgemeinen werden zu diesem Zweck Rechner zu Clustern zusammengeschaltet.

13.2 Verteilte Dateisysteme

Frage 13.7: Was versteht man unter einem verteilten Dateisystem?

Der Unterschied zu einem lokalen Dateisystem liegt darin, dass die Dateien über die verschiedenen Rechner bzw. Sites verteilt sein können. Ein verteiltes Dateisystem sollte idealerweise dieselben Dienste wie ein lokales Dateisystem anbieten, wie es in Kapitel 10 beschrieben wurde. Dazu zählen z.B. das Anlegen einer Datei, das Löschen einer Datei und das Lesen und Schreiben in einer Datei.

Frage 13.8: Was versteht man unter Naming und welche Ansätze gibt es?

Naming ist die Abbildung zwischen logischen und physikalischen Objekten. In einem lokalen Dateisystem wird der logische Dateiname auf eine eindeutige Nummer abgebildet, welche wiederum auf physikalische Diskblöcke abgebildet wird. In einem verteilten Dateisystem kommt eine zusätzliche Dimension hinzu, nämlich der Ort im Netzwerk, an dem sich die Datei befindet.

Es kann zwischen zwischen zwei verschiedenen Arten der Abbildungen unterschieden werden:

Ortstransparenz: Der Name einer Datei lässt nicht darauf schließen, wo diese Datei gespeichert ist.

Ortsunabhängigkeit: Der Name einer Datei muss nicht geändert werden, wenn sich der Speicherort der Datei ändert.

Wie leicht ersichtlich ist, setzt Ortsunabhängigkeit die Ortstransparenz voraus. In der Praxis wird meist eine statische, ortstransparente Abbildung verwendet (z.B. NFS). In solchen Systemen ist keine Dateimigration möglich. Dateimigration bedeutet, dass das Betriebssystem unabhängig vom Benutzer Dateien zwischen den verschiedenen Rechnern verschiebt. Dies kann zum Beispiel benutzt werden, um Load Balancing zu betreiben.

Ortsunabhängigkeit ermöglicht einen höheren Grad an Abstraktion zwischen Daten und Speicherorten, da sie die Namenshierarchie von der Speichermedienhierarchie trennt.

Frage 13.9: Welche drei verschiedenen Schemata können beim Naming verwendet werden?

Es können drei verschiedene Ansätze unterschieden werden:

1. Die Dateinamen ergeben sich als eine Kombination von Speicherort und Dateinamen, z.B. *host:local-name*. Dieser Ansatz ist nicht transparent aber eindeutig.

2. Verzeichnisse, welche nicht auf dem lokalen Rechner liegen, werden in die lokale Verzeichnisstruktur eingemountet. Der Vorteil dieses Vorgehens ist, dass es transparent ist. Der Nachteil liegt darin, dass ein nicht lokales Verzeichnis auf verschiedenen Rechnern in unterschiedliche Verzeichnisse eingehängt werden kann, wodurch das System dem Benutzer gegenüber nicht einheitlich erscheint.

3. Eine globale Verzeichnisstruktur enthält die Dateien aller im System enthaltenen Rechner. Das System erscheint für den Benutzer transparent und einheitlich, egal welche Maschine benutzt wird. Der Nachteil ist die schwere Implementierbarkeit, die sich z.B. daraus ergibt, dass es in vielen Systemen Spezialdateien gibt (z.B. `/dev/...`).

Frage 13.10: Wie funktioniert entfernter Dateizugriff mit Caching? Warum wird Caching betrieben?

Caching wird benutzt, um den Verkehr im Netzwerk zu begrenzen und den Zugriff auf eine Datei zu beschleunigen. Würde kein Caching benutzt werden, so müsste jeder Zugriff auf eine Datei explizit über das Netzwerk zu einem Rechner geleitet werden, welcher die Anfrage bedient. Diese Vorgehensweise wird *remote-service* genannt.

Beim Caching wird bei einer Anfrage nachgeschaut, ob sich das Ergebnis dieser Anfrage schon im Cache befindet. Ist dies der Fall, so werden die Daten im Cache benutzt, ansonsten werden die Daten über das Netzwerk übertragen und in den Cache eingetragen. Es existieren dann mindestens zwei Versionen der Datei (bzw. des Teils der Datei). Die Master-Kopie auf dem Server und die lokalen Kopien in den einzelnen Caches. Es ergeben sich nun drei Fragen:

- Wo werden die Daten gecachet?
- Wie wird die Master-Kopie aktualisiert?
- Wie werden die einzelnen Caches konsistent gehalten?

Frage 13.11: Welche Möglichkeiten gibt es, die Master-Kopie zu aktualisieren?

Es kann zwischen den folgenden Möglichkeiten unterschieden werden:

write-through: Sobald eine Änderung an der lokalen Datei vorgenommen wird, wird diese an die Master-Kopie weitergegeben. Vorteil dieses Vorgehens ist ein erhöhte Verlässlichkeit. Nachteilig ist bei diesem Vorgehen, dass viele kleine Schreibzugriffe benötigt werden.

delayed-write: Änderungen werden erst nur im lokalen Cache vorgenommen und zu späteren Zeitpunkten an die Master-Kopie geschickt. Dabei gibt es verschiedene Strategien, zu welchem Zeitpunkt dies geschieht:

- Bevor ein Block aus dem Cache entfernt wird, wird geschaut, ob er modifiziert wurde. Ist dies der Fall, wird die Master-Kopie informiert. Vorteil: Wegen der Entfernung des Blocks ist sowieso eine I/O-Operation notwendig, also kann dies kombiniert werden. Nachteil: Blöcke können sehr lange im Cache bleiben.
- Anschauen aller Blöcke in regelmäßigen Abständen, z.B. alle 30 Sekunden.

Der Vorteil liegt darin, dass Schreibvorgänge sehr schnell gehen und das Teile der Datei, die in kurzen Abständen mehrfach überschrieben werden, nicht jedes Mal an den Server geschickt werden müssen. Das Verfahren des delayed-write hat jedoch den Nachteil, dass im Falle eines Crashs viele Daten verloren gehen können.

write-on-close: Die Änderungen werden erst beim Schließen der Datei an die Master-Kopie mitgeteilt. Dies kann den Schließvorgang etwas verzögern, hat jedoch den Vorteil, dass es für Dateien, die lange geöffnet bleiben, einen Performancegewinn bedeutet. Werden Dateien jedoch andererseits oft kurz geöffnet und geschlossen, so wirkt sich dies eher negativ aus.

Frage 13.12: Welche Möglichkeiten gibt es, die Konsistenz in einem verteilten Dateisystem zu bewahren?

Wenn eine Kopie aus dem Cache benutzt wird, muss dafür gesorgt werden, dass diese stets konsistent mit der Master-Kopie auf dem Datei-Server ist. Dieses Problem wird auch Cache-Konsistenz-Problem genannt. Ist die Kopie im Cache nicht mehr konsistent, so muss dafür gesorgt werden, dass sie es wieder wird. Dazu gibt es zwei Ansätze:

Vom Client initiiert: Der Client fragt beim Server an, ob seine Kopie noch aktuell ist. Dies kann, wie die Aktualisierung der Master-Kopie, zu verschiedenen Zeitpunkten geschehen. Es kann z.B. nach jeder Änderung ein Konsistenzcheck durchgeführt werden, nur beim Öffnen der Datei oder in regelmäßigen Abständen.

Vom Server initiiert: Der Server merkt sich für jeden Client, welche Dateien dieser geöffnet hat und welche Teile dieser Dateien beim Client gecachet sind. Haben mindestens zwei Clients eine Datei in der Art und Weise geöffnet, dass ein Konflikt entstehen kann, so muss der Server handeln. In einem solchen Fall kann eine Lösung z.B. sein, den Clients mitzuteilen, dass die Datei nicht gecachet werden darf.

Frage 13.13: Was ist der Unterschied zwischen stateful und stateless Service?

Ist der Dateiserver **stateful**, also zustandsbehaftet, so wird sich für jeden Client gemerkt, welche Dateien er geöffnet hat. Logisch gesehen, besteht eine

dauerhafte Verbindung zwischen Client und Server, solange eine Datei geöffnet ist. Diese Verbindung wird über eine eindeutige Identifikationsnummer geregelt, d.h. der Client schickt diese Nummer mit jeder Anfrage mit. Diese Identifikationsnummer funktioniert dabei quasi wie ein Datei-Deskriptor, d.h. der Server lädt die benötigten Informationen der Datei in den Hauptspeicher und bedient Anfragen hieraus. Wird die Verbindung getrennt, so muss dieser Speicher wieder freigegeben werden. In diesem Zusammenhang ist es auch nötig, Abstürze eines Clients zu erkennen und Garbage Collection zu betreiben. Der Absturz des Servers ist recht problematisch, da sehr viele Informationen verloren gehen, welche z.B. durch Kommunikation mit den Clients erst mühsam wieder gewonnen werden muss.

Ist der Dateiserver **stateless**, so werden serverseitig keine Statusinformationen gespeichert und jede Anfrage ist in sich abgeschlossen. Diese Methode kommt problemlos mit Abstürzen des Servers oder des Clients klar, ist allerdings auch ineffizienter, da die Anfragepakete größer sind und die Bearbeitung einer Anfrage ebenfalls länger dauert.

Frage 13.14: Was versteht man unter Replika und was sind ihre Vorteile?

Um die Verfügbarkeit einer Datei zu erhöhen, können Dateien in Form von Replikas redundant abgespeichert werden. Dies sollte für den Benutzer transparent geschehen, d.h. ein Dateiname kann auf mehrere Replikas verweisen. Auf einer tieferen Ebene im System müssen die Replikas jedoch unterscheidbar sein. Es ist in einigen Fällen ebenfalls wünschenswert, den Grad der Redundanz durch den Benutzer beeinflussbar zu machen.

Andererseits ergibt sich hier nun das Problem, dass die Replikas untereinander konsistent gehalten werden müssen. Dies bedeutet, dass wenn ein Benutzer eine Datei ändert, diese Änderungen in allen Replikas sichtbar sein müssen.

Abbildungsverzeichnis

1 Die einzelnen Komponenten eines Computersystems 4
2 Speicherlayout eines einfachen Batch-Systems. 6
3 Struktur eines Computersystems 9
4 Die fünf Zustände, in denen sich ein Prozess befinden kann . . . 13
5 Inhalt des Process Control Block (PCB) 14
6 Ready und I/O-Queue 15
7 Mehrere Threads in einem Prozess 22
8 Histogramm von CPU Bursts 25
9 FCFS-Scheduling bei der Reihenfolge P_1, P_2 und P_3 28
10 FCFS-Scheduling bei der Reihenfolge P_2, P_3 und P_1 28
11 SJF-Scheduling 29
12 Preemptives SJF Scheduling 30
13 Priority Scheduling 31
14 Round Robin-Scheduling 32
15 Beispiel für ein RAG (in einem Deadlock-Zustand) 50
16 RAG mit einem Kreis ohne Deadlock 51
17 Die verschiedenen Zustände, in denen sich ein System befinden kann 54
18 Deadlock Detection mit Resourcen, die nur eine Instanz haben . 57
19 Overlays für einen 2-Pass-Assembler 62
20 Relokierung mit einem Relocation Register 63
21 Relokierung mit einem Limit und einem Relocation Register . . . 64
22 Modell des logischen und physikalischen Speichers beim Paging . 66
23 Grundlegende Idee des Paging 66
24 Die Aufteilung einer logischen Adresse beim Paging 67
25 Verwaltung und Zuweisung freier Frames 67
26 Grundlegende Idee des Paging 69
27 Eine gepagete Pagetabelle 70
28 Grundlegende Idee des Paging 70
29 Prinzip einer invertierten Pagetabelle 71
30 Benutzerperspektive auf ein Programm 72
31 Abbildung der logischen auf die physikalischen Adressen beim Segmentieren 73
32 Beispiel für Segmentierung 74
33 Beispiel für das Working Set Modell 84
34 Zusammenhang zwischen der Häufigkeit der Page Faults und Thrashing 84
35 Zusammenhängende Allokation 95
36 Verkettete Allokation 97
37 Prinzip eines FAT 98
38 Indizierte Allokation 98
39 Struktur einer Inode 99
40 Verwaltung von freien Blöcken mit einer verketteten Liste 100

Liste der Algorithmen

1 Der Bakery-Algorithmus für Prozess P_i . . . 39
2 Critical Section ohne test-and-set . . . 41
3 Critical Section mit test-and-set . . . 41
4 Critical Section mit swap . . . 41
5 Implementierung der P()-Operation . . . 44
6 Implementierung der V()-Operation . . . 45
7 Der Code des Produzenten . . . 46
8 Der Code des Kosumenten . . . 46
9 Der Code eines Writers . . . 46
10 Der Code eines Readers . . . 47

Literatur

[Sil98] SILBERSCHATZ, Abraham: *Operating System Concepts*. Fünfte Auflage. Addison-Wesley, 1998

[Sil00] SILBERSCHATZ, Abraham: *Applied Operating System Concepts*. Erste Auflage. Wiley, 2000

[Tan95] TANENBAUM, Andrew S.: *Moderne Betriebssysteme*. Zweite Auflage. Hanser, 1995