OSVO Zusammenfassung

Smonman

1. März 2022

Inhaltsverzeichnis

1 Einführung

1.1 Historische Entwicklung

Entwicklung vom Betriebssystem (BS) ist sehr stark mit der Entwicklung der Hardware verschränkt. **Wechselseitige Abhängigkeit zwischen Betriebssystem und Hardware** fordern die Weiterentwicklung.
Früher waren die Rechner sehr teuer, jedes Gerät wurde handgefertig und nur von Spezialisten bedient → Rechenzeit war teuer
Im Laufe der Zeit wurde die Hardware immer billiger, heutzutage kann sich jeder einen Computer leisten. Arbeitskräfte sind teurer als Computer → Computer entlasten und ersetzten Arbeitskräfte

1.1.1 Serial Processing (1950)

Kein Betriebssystem → **Programmierung direkt an der Hardware**
I/O Elemente bestanden aus:

- Schalter
- Lampen
- Kartenleser
- Drucker

Kein interaktiver Betrieb möglich; Informationen über Programme mussten von Spezialisten verwaltet werden (wo steht was im Speicher usw.)
Die Art einen Computer so zu verwenden ist sehr fehleranfällig → Zusammenfassen von dem was jedes Programm braucht. Es entstanden die ersten **Unterprogrammbibliotheken** für I/O; gut getestete einzelne Bibliothek, als Arbeits- und Fehlerersparnis.
Neben dem einstellen der I/O benötigt auch das Setup von Programmen viel Zeit. Um dieses Problem zu lösen gab es zwei Ansätze:

- **Spezialisten anstellen**: Ein trainierter Arbeiter erhält die Lochkarten von diversen Programmen, welche ausgeführt werden sollen. Der Computer wird vor jedem Programm neu aufgesetzt, das Programm ausgeführt, die Ergebnisse eingesammelt und abgelegt. Danach können die verschiedenen Benutzer ihre Ergebnisse abholen kommen.
- **Monitor**: Die andere Möglichkeit wäre es, genau die Arbeit eines spezialisierten Arbeiters auch zu übernehmen; das Einstellen des Computers, starten des Programms und Entnahme der Ausgabe können genauso automatisiert werden, wenn diese Aktionen in eine Befehlsprache codiert werden kann

1.1.2 Monitor (1960)

Mithilfe eines Monitors können verschiedene Jobs abgearbeitet werden. Dafür wurde die Job Control Language (JCL) eingeführt; Sequenzielle Abarbeitung aller Jobs in einer Schleife

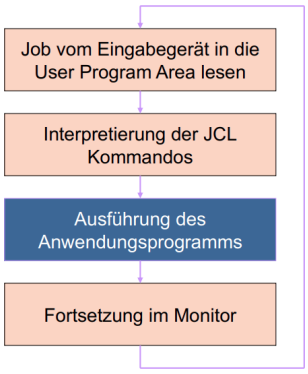

Abbildung 1: Job-Ausführung mit Monitor

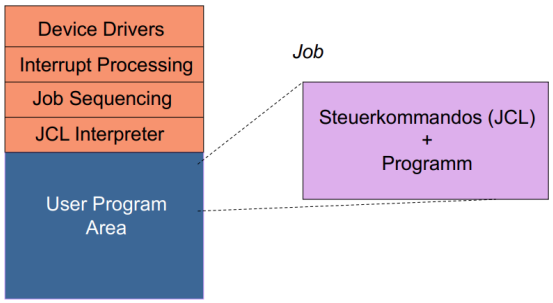

Abbildung 2: Monitor

Aufbau eines Monitors

- **Device Driver**: Nachfolger der I/O Routinen, können bestimmte I/O Geräte ansteuern und verwenden
- **Interrupt Processing**: Nachdem die verschiedenen Jobs in einer Schleife abarbeitet werden, muss es ein System geben, welches bei einem Fehler den Ausstieg aus der Schleife ermöglicht → Interrupts
- **Job-Sequencing**
- **JCL**: Job Control Language; ermöglicht das Codieren von den Setup-Aufgaben; Außerdem wird ein Interpreter für die JCL benötigt
- **User Program Area**: Hier werden die tatsächlichen Programm hineingeladen und ausgeführt
- **Job**: Ein Job besteht aus dem Programm sowie des Steuerkommandos, welches in JCL geschrieben ist, um die Maschine vorzubereiten und das Programm zu starten

Anforderungen an die Hardware Durch die Entwicklung des Monitor Prinzips, wurden auch weitere Anforderungen an die Hardware gestellt:

- **Speicherschutz**: Es musste sichergestellt werden, dass ein Programm nicht auf den Speicher des Monitors zugreifen darf; Wenn ein Fehler in einem Programm passierte, so durfte dadurch nicht das Monitor Programm beeinträchtigt werden
- **Timer**: Hardware Timer können Limitierung der Rechenzeiten eingesetzt werden. Wenn z.B. Ein Programm über einen gewissen Zeitraum nicht fertig ist, soll dieses Unterbrochen werden.
- **Interrupt**: Dafür sind Interrupt-Leitungen notwendig, sowie ein Prozessor, welcher Interrupts verwerten und sogenannte Interrupt-Service-Routinen ausführen kann
- **Privilegierte Instruktionen**: Das Monitor Programm sollte über privilegierte Instruktionen verfügen, welches nicht von Programmen aus dem User Program Area verwendet werden können, wie z.B.:
  - Laden und Ausführen von Programmen
  - Speicher und Lesezugriff auf bestimme Speicherabschnitte

Ein Monitor hat mittlerweile schon Ähnlichkeiten mit einem Betriebssystem wie es heute gekannt wird. Das nächste Problem welches erkannt wird ist die **ungleichmäßige Auslastung des Computers**. Es werden entweder die I/O-Systeme oder die Rechenleistung benötigt, allerdings selten beides. Da dadurch teure Rechenzeit verloren geht will man dieses Problem lösen → Parallele

CPU und I/O Aktivität. Das kann durch **Pufferung der I/O Aktivitäten** erzielt werden → Drucker lesen z.B. selbständig parallel aus einem Puffer aus, oder Eingabe wird zuerst in einen Puffer geschrieben; CPU kann währenddessen für etwas anderes verwendet werden.

1.1.3 Multiprogramming (1965)

Sehr oft lassen sich die Programme in zwei Arten Aufteilen:

- **CPU bound**: CPU-Intensive Programme
- **I/O bound**: I/O-Intensive Programme

Um die CPU gleichmäßiger auszulasten, und dadurch an Rechenzeit zu gewinnen, entwickelt man das **Multiprogramming (Multitasking)** Prinzip:

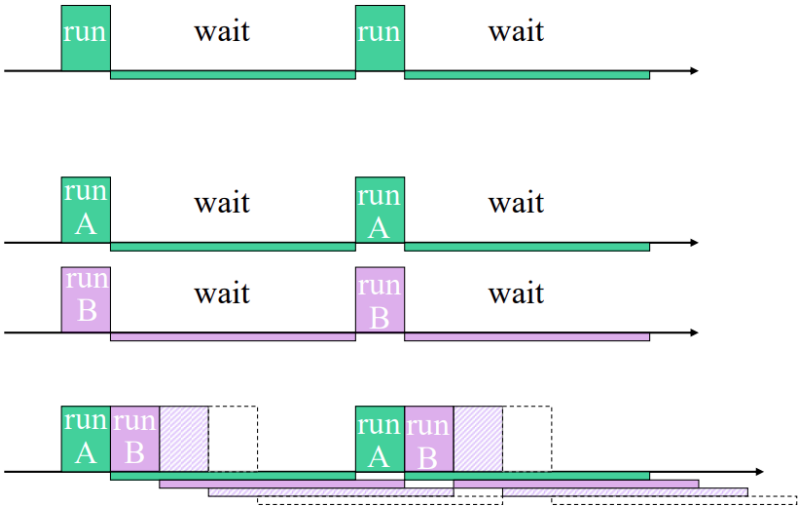

Abbildung 3: Multiprogrammed Batch Systems

Durch dieses neue Prinzip werden allerdings wieder neue Mechanismen notwendig:

- **bessere Speicherverwaltung**: Die Speicherverwaltung muss jetzt nicht mehr nur zwischen Programm und Monitor gelöst werden, sondern auch zwischen Programmen; ein Programm darf nicht in den Speicher eines anderen Programms schreiben oder davon lesen
- **Verwalung von CPU und Ressources**: Plötzlich ist die Verwaltung der CPU und Ressourcen ein Thema (**Scheduling**):
  - Wo werden Programme unterbrochen?
  - Welche Programme habe ich überhaupt?
  - Welche I/O Daten gehören welchem Programm?

1.1.4 Time-Sharing Systems (1970)

ca. 1970 sind Computer bereits deutlich billiger geworden und sollen teure Arbeitskräfte ersetzten → mehrere Benutzer teilen sich ein Gerät
Jeder Benutzer soll sich fühlen, als wäre er der einzige Benutzer des Systems. Neue Ideen werden Entwickelt:

- **Preemptive Scheduling**
- **Prioritäten**

Dadurch dass der Computer jetzt von vielen Menschen zu bedienen ist, und nicht mehr nur von Spezialisten, soll er für einen Benutzer leichter bedienbar gemacht werden → Programme und Daten sollen leicht verfügbar gemacht werden
Um das zu erreichen wird ein **Filesystem** dem Rechner hinzugefügt. Später kommen dann auch noch fest verbaute Speichermedien hinzu, die Festplatte.

1.2 Weitere Entwicklung

- Immer billigere Hardware → jeder User hat einen eigenen Rechner
- Computer ziehen in große Büros ein:
  - GUIs, vereinfacht die Nutzung
  - Networking
  - Security, als Konsequenz von Networking
- Wachstum des Internets
- Mobilität (z.B. Handy)
- Mehrere Peripheriegeräte (Voice, Audio, Video)
- Embedded Computing, Internet of Things (Real-Time...)
- Multicores, erleichtern und erschweren Parallelisiserung zugleich

1.3 Was ist ein Betriebsystem?

- **Interace**: Schnittstelle des Computers dahinter, erleichter Verwendung und ermöglicht Nutzung von mehreren (ungeschulten) Personen
- **Virtuelle Maschine**: ganze Programme können als Instruktionen eines Betriebssystems gesehen werden
- **Dienstleister**
  - **Programmausführung**
  - **Programmerstellung**
  - **I/O, Filezugriff, Netzwerkkommunikation**
  - **Zugangskontrolle**
  - **Fehlererkennung und Behandlung der Maschine**
  - **Logging (Überwachung)**
- **Ressourcenmanager**
  - **keine *externe* Kontrollinstanz**: Das Betriebssystem ist alleinig Verantwortlich
  - **Ressourcenmanaging**: Verwendet selber Ressourcen, übergibt allerdings die Kontrolle und Ressourcen, kontrolliert an verschiedene Programme ab. Das zurückbekommen der Kontrolle ist essentiell und mitunter auch eine Security-Frage
  - **Funktionalität**: Hebt sich durch die Funktionalität von *herkömmlichen* Programmen ab; Leitet den Prozessor bei der Verwendung der Ressourcen und Zeitzuteilung
- **Abstraktion**: Betriebssystem abstrahiert den Computer und gaukelt dem Benutzer etwas vor:
  - **Ungestörte Programmabarbeitung**: Jedes Programm fühlt sich so an, als wäre es das einzige, welches auf der Maschine laufen würde
  - **Unendlich großer Speicher**: Betriebssystem lädt nur die wirklich notwendigen Daten in den RAM → es können mehr Programme verwendet werden als von einem User angenommen; User muss sich keine Sorgen machen ob man ein Programm überhaupt starten kann (ob es genug freien RAM gibt)
  - **Private Maschine**: Jeder Benutzer sieht nur sein Zeug auf der Maschine, für jeden Benutzer fühlt es sich an, als wäre er der einzige Benutzer; Datensicherheit

So gesehen macht ein Betriebssystem einen Computer erst verwendbar.

2 Prozesse

Ein Prozess ist das Programm in der Ausführung. Während der Programmcode statisch ist, ist ein Prozess dynamisch. Vergleichbar mit einem Zug (statisch) und der Zugfahrt mit diesem Zug (dynamisch)
Durch einen **Prozess ändert** sich der **Zustand eines Computers** (Stack, Puffer, Variablen usw.), das bedeutet ein Prozess beschreibt die dynamischen Aspekte eines Programms.
Kontext eines Prozesses:

- **Aktueller Zustand** des Prozesses (Process Counter, Register, usw.)
- **Daten zur Prozessverwaltung** (Wartebedingungen, Priorität, usw.)

2.1 Prozesse im Betriebssystem

Normalerweise kann immer nur ein Prozess auf dem Prozessor laufen. (In diesem Fall gehen wir von einem Prozessor mit einem Kern aus) Das bedeutet ein Dispatcher muss den Prozessanlauf und -tausch organisieren. Der Dispatcher ist wie ein eigener Prozess, welcher vom Betriebssystem gesteuert wird.

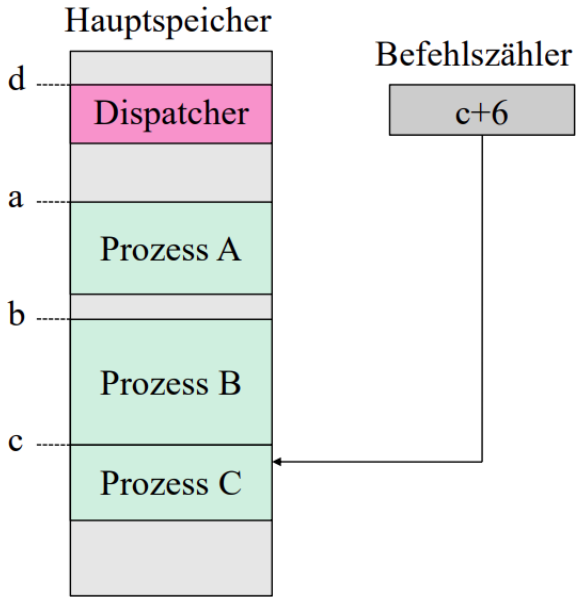

Abbildung 4: Prozesse im Betriebssystem

2.1.1 Trace

Ein Trace charakterisiert das Verhalten eines Prozesses. Es ist die **Sequenz an Instruktionen**, welche für einen Prozess ausgeführt werden. Mehrere Traces, bzw. die Überlappung davon charakterisiert das Prozessorverhalten: Auf der rechten Seite (6) kann ein Trace über mehrere Prozesse

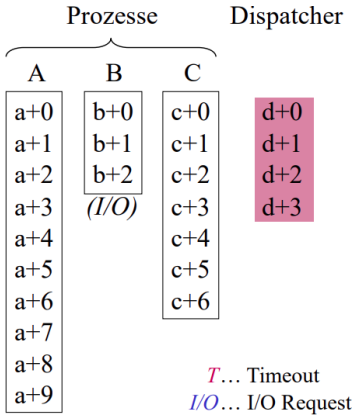

Abbildung 5: Prozesse und Dispatcher

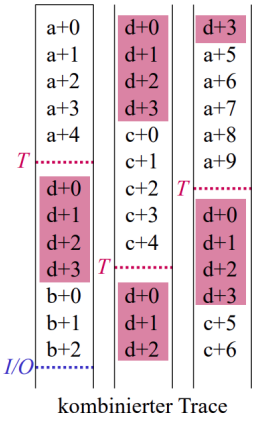

Abbildung 6: Kombinierter Trace

gesehen werden. Das Betriebssystem teilt die Prozessorzeit genau ein. Damit werden Prozesse *in Schach gehalten*, um nicht unverhältnismäßig viel Rechenzeit zu verbrauchen. Bei einer I/O Request, wird meist der nächste Prozess ausgeführt, da diese für einen Computer sehr viel Zeit braucht.
Was hier auch gut zu sehen ist, wie **parallele Ausführung durch schnelle abwechselnde sequenzielle Ausführung simuliert** wird.

2.2 Einfachstes Prozessmodell

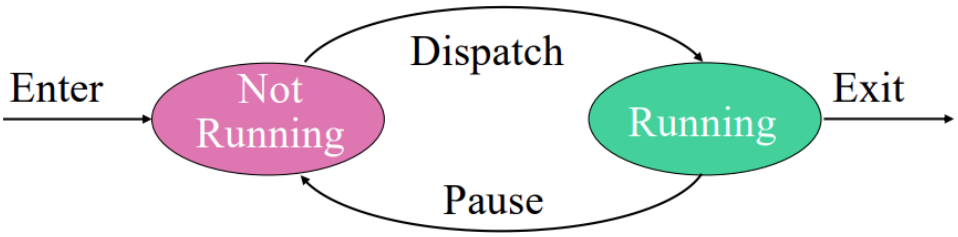

Abbildung 7: Einfaches Zustands-Prozessmodell

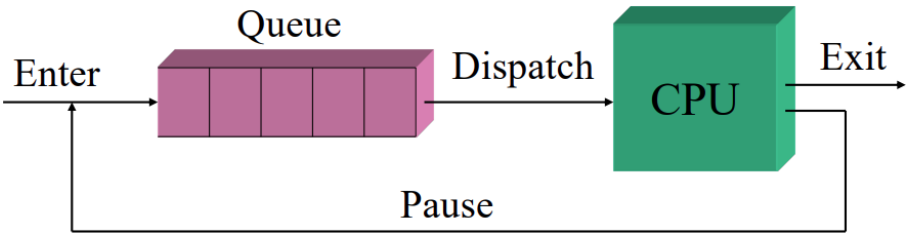

Abbildung 8: Einfaches Queue Modell

Bei dem einfachsten Prozessmodell gibt es bloß die Zustände *Not Running* und *Running*. Damit das Betriebssystem weiß welche Prozesse weiß welche Prozesse es gibt, werden die, die gerade nicht laufen in einer Queue gespeichert. Der Dispatcher kann dann Prozesse aus dieser Queue entnehmen und der CPU zuteilen. Wie der Dispatcher entscheidet, welcher Prozess ausgewählt wird, ist unterschiedlich. Allerdings sind Prozesse ja nicht nur in einen der beiden Zuständen. Sie können erzeugt und beendet werden.

2.2.1 Erzeugung von Prozessen

Bei der Erzeugung eines Prozesses werden bestimmte **Datenstrukturen für einen Prozess** angelegt. Um diese auch abspeichern zu können **allokiert das Betriebssystem genügend Speicher**. Jedem Prozess wird eine **Prozessnummer** (PID) zugeteilt, außerdem wird zur Prozesstabelle ein Eintrag für den neuen Prozess hinzugefügt. Diese **Prozesstabelle** ist für die Prozessverwaltung notwendig.
Nach der Erzeugung eines Prozesses ist dieser allerdings noch nicht direkt laufbereit. Das vermeidet Ressourcenüberlastung durch das Zulassen zu vieler Prozesse auf einmal.
Wann wird ein Prozess erzeugt?

- Login eines interaktiven Benutzers
- **Betriebssystem**: Ausführung eines Services → ein Betriebssystem kann für seine eigene Funktionalität Prozesse erzeugen
- **Benutzer**: Erzeugung durch eines Benutzerprozesses (`fork()`), auch *Prozessspawning* genannt.
- Absetzen eines neuen Jobs

2.2.2 Beendigung von Prozessen

Bevor die zugehörigen Informationen zu einem Prozess nach der Beendigung gelöscht werden werden diese noch womöglich von Hilfsprogrammen verwendet (z.B. Debugging, Logging) Danach werden die Datenstrukturen gelöscht, und der dazugehörige Eintrag in der Prozesstabelle entfernt. Generell werden **alle zum Prozess gehörenden Informationen gelöscht**.
Wann werden Prozesse beendet?

- Logout durch Benutzer
- Service Request an das Betriebssystem
- Auftreten eines Fehlers bei der Abarbeitung eiens Prozesses
- Halt-Instruktion eines Jobs

2.3 Prozesszustände

Wesentliche Prozesszustände:

- **Running**: Prozess ist im Besitz der CPU und wird ausgeführt
- **Ready**: Prozess ist laufbereit, wartet allerdings auf die Zuteilung zur CPU
- **Blocked / Waiting**: Prozess ist nicht laufbereit, und wartet auf ein Ereignis (z.B. I/O Aktionen, Timer Ereignisse bei Weckern)

2.4 5 Zustands-Prozessmodell

Aus diesen Prozesszuständen sowie dem Erzeugen und Beenden von Prozessen ergibt sich dann das 5 Zustands-Prozessmodell Dieses Prozessmodell wird durch zwei weitere Zustände ergänzt:

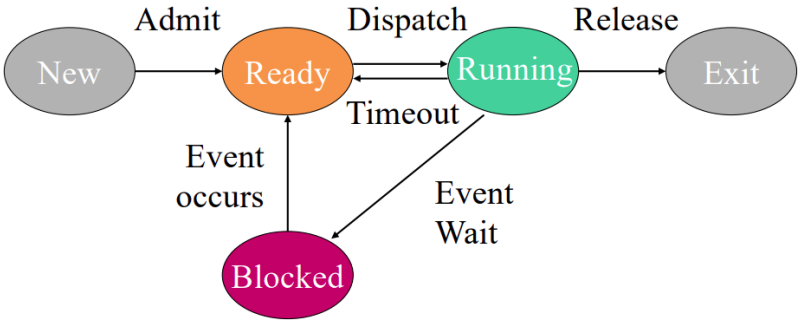

Abbildung 9: 5 Zustands-Prozessmodell

- **New**: Das Betriebssystem hat einen neuen Prozess erzeugt
- **Exit**: Dieser Zustand wird durch das Terminieren eines Prozesses erreicht. Der Prozess wird nicht weiter ausgeführt

Passend zu dem *Blocked* Zustand, in dem Prozesse auf ein bestimmtes Ereignis warten, gibt es verschiedene Event-Queues für bestimmte Ereignisse. Sobald ein Ereigniss eintritt werden die Prozesse, welche in der korrespondierenden Queue warten, in die Ready-Queue verschoben.

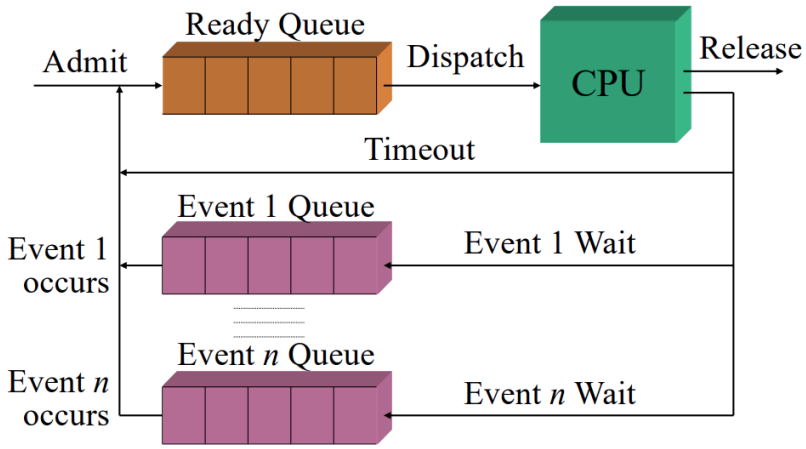

Abbildung 10: Queueing Prozessmodell

2.5 Process Switch

- **Supervisor Call** (System Call): Prozess nimmt einen Service vom BS in Anspruch. BS kann die Kontrolle übernehmen und danach diesen Prozess direkt wieder aufrufen oder einen anderen Prozess starten. Je nach dem wie lange der Service vom BS dauert.
- **Trap**: z.B. Fehler in einem Prozess → Verzweigung auf eine bestimmte Adresse → Betriebssystem ist in Gewalt
- **Interrupt**: Ursache von Außen (z.B. Hotkeys), Prozessor muss diese unterstützen (Interrupt-Leitungen) → BS unter Kontrolle und kann einen Process Switch 2.5 durchführen.

2.6 Swapping

Wenn zu viele Prozesse im RAM untergebracht sind, und der Speicher zu wenig wird, können Prozesse in den Hintergrundspeicher geschrieben werden.
Zu viele Prozesse im Hauptspeicher führen zu einer schlechteren Performance → Auslagerung von Prozessen
Um dieses Prinzing allerdings in einem Betriebssystem unterbringen zu können, braucht es zwei neue Zustände in dem Prozessmodell:

- **Ready suspend**
- **Blocked suspend**

Ausgelagerte Prozesse sind *suspendiert*. Ob ein Prozess ausgelagert werden soll, entscheidet das Betriebssystem
Es können Prozesse, welche im Ready oder im Blocked Zustand sind ausgelagert werden. Außerdem können Prozesse direkt in den Ready suspend Zustand *admittet* werden.
Ursachen für ein Suspend:

- **Swapping**
- **Problembehafteter Prozess auslagern**
- **Interaktive Request**: z.B. Debugging
- **Timing**: periodischer Prozess kann zwischen den Aktivierungen ausgelagert werden. z.B: Wecker
- **Parent Request**: Untersuchung, Modifikation, Koordination von Kindprozessen

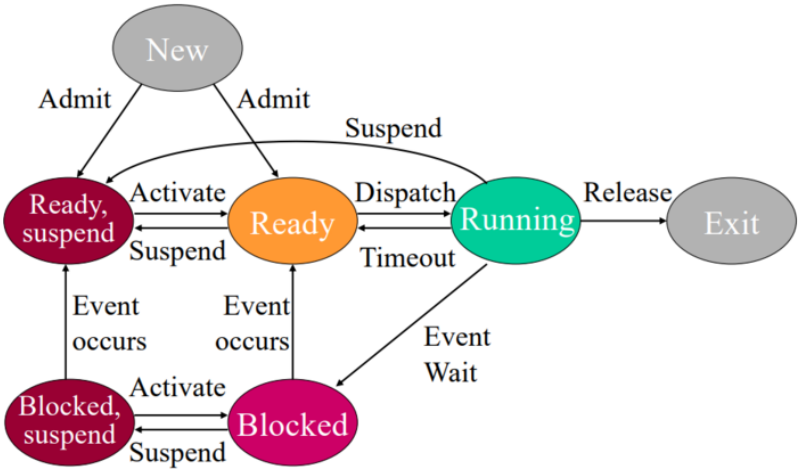

Abbildung 11: Erweitertes 5 Zustands-Prozessmodell

2.7 Kontrollstrukturen für die Prozessverwaltung

Was braucht das Betriebssystem für Datenstrukturen, um Prozesse verwalten zu können?
Das Betriebssystem verwaltet folgende Tabellen für Prozesse und Ressourcen:

- **Memory Tables**: Wo befindet sich Code und Daten für den Prozess im Arbeitsspeicher?
- **I/O Tables**:
  - Welche Geräte verwendet ein Prozess?
  - In welchen Zustand sind die I/O Geräte?
  - Gibt es gepufferte Daten?
  - Gibt es Daten, die zu übertragen sind?
  - Sind Daten für einen Prozess angekommen?
- **File Tables**:
  - Welche Dateien sind offen?
  - Wo bin ich gerade beim Lesen einer Datei? (Wo ist der Cursor?)
  - Welche Rechte haben diese Dateien?
- **Prozess Tables**: Verweise auf ein Process Image

2.8 Process Image

Essentielle Daten für einen Prozess:

- **Code für das Programm**
- **Veränderliche Daten**: Alles was sich im Verlauf der Abarbeitung verändern kann. Modifizierbarer Bereich des User Space, User Stack (Stack im User Space, ist nicht derselbe Stack wie für System Calls usw.)
- **System Stack**: Parameter für Calling Address von System Calls, speziell geschützt, damit der User nicht mit dem Betriebssystem interferrieren kann
- **Process Control Block (PCB)** (Execution Context):
  - Process identification (Prozessnummer, PID), Information über den Parent Process
  - Processor state information
  - Process control information: Angaben zum Process die für die Verwaltung notwendig sind

  Kerninformationen des Prozessors; Hauptverwaltungsstruktur

Der **Process Control Block (PCB) wird immer im Arbeitsspeicher gebraucht**, das gesamte Process Image hingegen nicht, es befindet sich im virtuellen Speicher.

2.8.1 PCB: Process Identification

- **Eindeutige Prozessnummer (Process Identifyer)**: Index im Primary Process Table
- **Benutzerkennung (User Identifyer)**: Benutzer dem der Prozess gehört → Privilegien und Rechte leiten sich davon ab
- **Nummer des Elternprozesses (Parent Process Identifyer)**: Nummer von dem Prozess, der diesen gestartet hat. Wenn ein Prozess beendet wird, wird das dem Elternprozess mitgeteilt. → Elternprozess wartet auf Kinderprozesse und terminiert in der Regel wenn alle Kinderprozesse beendet sind → Übersicht über alle Prozesse wird erhalten; (Typischerweise hat man auch Verweise von den Elternprozessen auf die Kindprozesse)

2.8.2 PCB: Processor State Information

- **Registerinhalte**
- **Kontroll- und Statusregister**:
  - **Befehlszähler**
  - **Program Status Word (PSW)**: Kontroll- und Modusinformation, Status-Bits
- **Stack Pointer**

All diese Informationen sind normalerweise auf dem Prozessor erhältlich; Sobald ein Prozessor unterbrochen wird, muss der Zustand des Prozessors abgespeichert werden, damit der Prozess später reibungslos wieder anlaufen kann.

2.8.3 PCB: Process Control Information

- **Scheduling und Zustandsinformation**
  - Zustand in dem sich der Prozess befindet (Zustandsmodell)
  - Priorität und Schedulinginformationen
  - Ereignistyp auf den der Prozess wartet
- **Querverweise auf andere Prozesse**
  - Realisierung von Process-Queues (I/O Queues, Synchonisierungs-Queues)
  - Verweis auf Parent, Child ...
- **Interprozesskommunikation (IPC)**
  - Flags
  - Signale, Verweise auf Signalhandlern
  - Verweise auf Nachrichten
- **Privileges**: Was darf der Prozess? Welche Datein darf der Prozess öffnen? (Kann auch bei der Datei abgespeichert werden)
- **Memory Management**
  - Verweise auf Segmenttabellen
  - Verweise auf Seitentabellen
  - Wo ist was abgespeichert?
- **Ressourcen**
  - **Verwendete Ressourcen**: geöffnete Files, I/O Geräte
  - **Bisher konsumierte Ressourcen**: CPU Zeit, I/O, usw.

2.9 Prozesslisten

Diese **Prozess Control Blocks** werden **in den verschiedenen Queues** (Ready Queue, Blocked Queue) *aufgefädelt*, in Form einer veränderlichen Datenstruktur. Man muss natürlich darauf achten, dass nur das Betriebssystem darauf Zugriff hat. Die Instruktionen, bestimmte Prozesse (also eigentlich die PCBs) in eine Queue zu stecken, sind privilegierte Instruktionen die besonders geschützt werden müssen. Benutzerprozesse dürfen darauf keinen Zugriff haben.
Um herauszufinden, wann ein Prozess diese bestimmten Privilegien hat, oder als Betriebssystem läuft, gibt es Execution Modes 2.10.

2.10 Execution Modes

Um die Datenstrukturen des Betriebssystems zu schützen gibt es mindestens zwei Execution Modes (ein modernes Betriebssystem unterstützt eher drei bis vier):

- **Privileged Mode** (system mode, kernel mode, supervisor mode, control mode): Zugriff auf Kontrollregister, MM, I/O-Primitive; Zugriff auf die Datenstrukturen des Betriebssystems
- **User Mode**: Benutzercode und Prozesse laufen hier; Zugriff auf lokale Daten (User Space); kein Zugriff auf die Systemdaten (Shared Space)

Diese Modes werden in den **Mode-Bits** des Betriebssystems abgespeichert. Ein **Mode Switch** 2.10 ist die **Vorraussetzung für einen Process Switch** 2.5, da nur das Betriebssystem auf einen anderen Prozess wechseln kann. Auslöser für ein Mode Switch sind folgende Ereignisklassen:

- System Call
- Trap
- Interrupt

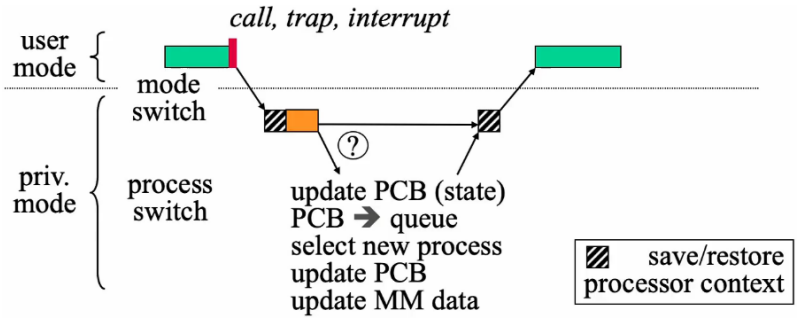

Abbildung 12: Mode Switch vs. Process Switch

Das Fragezeichen zeigt hier einen Process Switch 2.5 nach dem davor erfolgten Mode Switch 2.10 an.
Nicht jeder Mode Switch 2.10 bewirkt einen Process Switch! 2.5

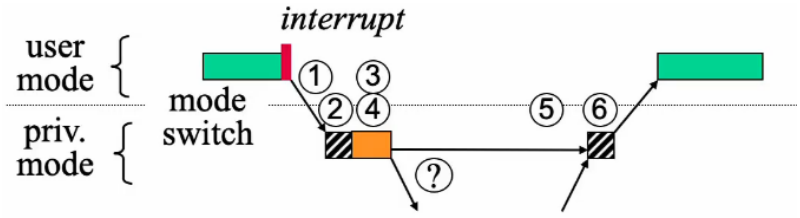

Abbildung 13: Mode Switch detailiert

1. PC → stack, ...; load PC from interrupt vector (Addresse der Interrupt Service Routine) [Hardware]
2. Save registers, setup new stack [assembly code]

3. Interrupt Service Routine, e.g. read and or buffer inputs [C code]
4. Scheduler decides on next process [C code]
5. Return to assembly code [C code]
6. Setup for process continuation [assembly code]

2.11 Prozess in Betriebssystemen

Es gibt verschiedene Möglichkeiten Betriebssysteme und Prozesse zu trennen:

- Strikte Trennung von Kernel und Prozessen
- Betriebssystem exekutiert innerhalb von Benutzerprozessen
- Prozessbasiertes Betriebssystem

2.11.1 Nonprocess Kernel

Strikte Trennung zwischen Prozess und Kernel → entweder im Prozess Kontext, oder im Kernel Kontext.
Call an das Betriebssystem → Verlassen des Prozess Kontexts, Betriebssystem *kennt* den zuletzt ausgeführten, und als nächstes kommenden Prozess nicht

- Prozessbegriff nur für Benutzerprogramme
- Verlassen des Prozesskontext bei Betriebssystem-Aktivität
- Betriebssystem arbeitet von Prozessen getrennt im Privileged Mode (eigener Speicherbereich, eigener Stack für Betriebssystem)

2.11.2 Betriebssystem-Ausführung in User-Prozessen

Gängiges Modell ist Betriebssystem Code in Prozess Kontext auszuführen (z.B. bei Mode Switch 2.10 vs. Process Switch 2.5 Erklärung)
Bei einem System Call wird der Kontext des Processes nicht verlassen → es gibt das Konzept eines *gerade jetzt laufenden Prozesses*
Betriebssystem Routinen kann sich auf den aktuellen Prozess beziehen (z.B. I/O Routinen), oder ein anderer Code sein → Betriebssystem läuft eigentlich nur im Kontext eines anderen Prozesses

- Betriebssystem ist eine Sammlung von Routinen, die vom Benutzerprogramm aufgerufen werden
- Fast alle Betriebssystem-Routinen laufen im Prozess Kontext
- Verlassen des Prozess Kontexts nur bei Process Switch 2.5

Die Datenstruktur ist in diesem Konzept etwas angepasst:

- Betriebssystem Code und Daten werden im Shared Address Space (= Speicherbereich den das Betriebssystem mit den Prozesse teilt, aber nur das Betriebssystem mit den Privilegien bearbeiten darf) abgelegt
- Getrennter Kernel Stack für Kernel Mode
- Im Prozess laufen sowohl User-Programm als auch Betriebssystem Routinen (→ Programm vs. Prozess)

Beim Mode Switch 2.10 wird in den Shared Address Space mit Privilegien verzweigt.

2.11.3 Prozessbasiertes Betriebssystem

- Betriebssystem ist eine Sammlung von Systemprozessen
- nur Basisservices sind nicht als Prozesse realisiert:
  - Process Switching 2.5
  - einfaches Memory Management
  - Interprocess Comunication (IPC)
  - Interrupts
  - I/O

Man versucht Systemaktivitäten in Prozessen zu realisieren.

2.12 Kernel-Architekturen

- **Monolithic Operating System**
  - Betriebssystem ist eine Menge an Prozeduren
  - Jede Prozeur kann jede andere Prozedur aufrufen
  - eher unübersichtlich / unstrukturiert
  - schwer zu garantieren, dass es keine Probleme enthält
- **Layered Operating System**
  - Hierarchische Organisation der Betriebssystem Funktionen
  - Interaktionen zwischen benachbarten Schichten
  - Mehrheit der Schichten exekutieren im Kernel Mode
  - Funktionalität kann strukturiert werden
  - Wechselwirkungen nur über die Grenzen der Schichten
  - Schichtungen nicht ganz intuitiv
  - Heutzutage eher Modulares Betriebssystem wo es nur im Low-Level Bereich Schichten gibt, worauf dann Module aufbauen

2.12.1 Microkernel Architektur

Architektur

- Basisservices im Kernel
- nicht zentrale Betriebssystem Services als Server Prozesse
- Nachrichtenkommunikation zwischen Betriebssystem und Prozessen
- Prozess Basiertes Betriebssystem
- Kernel soll so klein wie möglich sein, der Rest spielt sich in Prozessen ab

Services, die sich im Kernel abspielen müssen

- Prozess Switching
- Umschalten zwischen Basis- und Kernelprozessen
- Basic Memory Management (Strategie kann in einem Prozess realisiert werden, das tatsächliche Zuweisen aber im Kernel)
- Interrupts und Hardware Access (I/O)
- Nachrichtenaustausch und Nachrichtenkontrolle

Vorteile durch einen kleinen Kernel

- Einheitliche Interfaces
- Flexibilität und Erweiterbarkeit
- Portabilität
- Unterstützung von Verteilung
- leichter zu Analysieren und Debuggen

2.13 Zusammenfassung Prozess

- Prozess ist ein zentrales Konzept im Betriebssystem
- Betriebssystem kreiert, verwaltet und beendet Prozesse
- Prozess durchläuft verschiedene Zustände
- Datenstrukturen zur Prozessverwaltung:
  - Prozesstabelle
  - Process Image 2.8:
    - belegter Adressbereich
    - PCB 2.8: PID, Zustand, Steuerinformationen, Ressourcen, Priorität, usw.

2.14 Zusammenfassung Prozessor

- Mode Switch (zwischen User und Kernel Mode) 2.10
  - Interrupt
  - Trap
  - Supervisor Call (System Call)
- Execution Mode vs. Prozess / Betriebssystem Kontext
- Unterschiedliche Ansätze zur Implementierung von Betriebssystem und Prozessen:
  - strikte Trennung zwischen Betriebssystem und Prozesse
  - Betriebssystem-Funktionen in User Prozessen abgearbeitet
  - Server-Prozesse zur Ausführung von Betriebssystem Funktionen

3 Threads

Motivation ist das Entkoppeln von der Ressourcenverwaltung und der Ausführung → mehrere Threads pro Prozess (Multithreading). Prozess beinhaltet die Daten, ein Thread ist ein Ausführungsstrang eines Prozesses, lebt innerhalb eines Prozesses und verwendet die Ressourcen des Prozesses.

3.1 Prozesse und Threads

- **Prozess**
  - virtueller Adressraum mit Process Image 2.8
  - Speicherschutz, Files I/O Ressourcen
- **Thread**
  - Ausführungszustand (Running, Ready, ...)
  - Kontext (wenn nicht gerade laufend)
  - Stack
  - Thread-lokale statische und lokale Variablen
  - Zugriff auf Prozessspeicher und Ressourcen

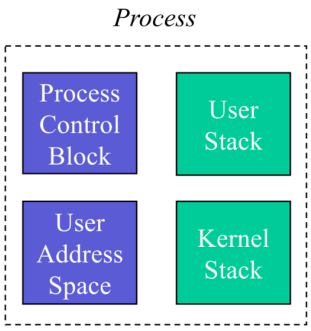

Abbildung 14: Singlethread Modell

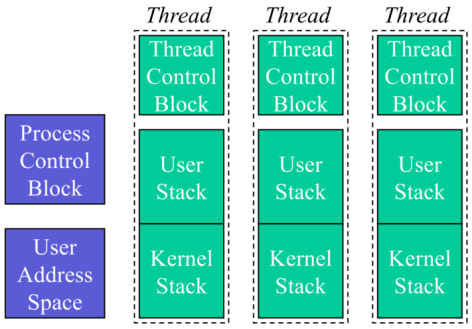

Abbildung 15: Multithread Modell

3.2 Vorteile von Threads vs. Prozesse

- Thread-Erzeugung benötigt weniger Zeit (Alle Daten sind schon durch die Prozess-Erzeugung bereitgestellt)
- Umschaltung zwischen Threads geht schneller als beim Process Switch 2.5
- Terminierung eines Threads benötigt weniger Zeit als die Prozessterminierung
- Kommunikation zwischen Threads eines Prozesses ohne Einschaltung des Kernels, aber **Synchronisation notwendig!**

3.3 Einsatzbereiche von Threads

Applikationen, die eine zusammengehörige Menge von Abarbeitungseinheiten bilden.

- Beispiel File Server in LAN:
  - mehrere Requests in kurzer Folge
  - ein Thread pro Request
- Beispiel Spreadsheet-Programm
  - ein Thread zeigt die Menüs an und liest Inputs
  - ein Thread führt Berechnungen und Updates aus

3.4 Thread Zustände

- Running
- Ready
- Blocked

Der Suspend Zustand existiert nicht für einzelne Threads, da der Suspend Zustand sich um die Ressourcenverwaltung eines Prozesses bezieht, aber alle Threads die Ressourcen des Prozesses verwenden. Man kann nicht einen einzelnen Thread auslagern, nur einen gesamten Prozess. Sobald ein Prozess terminiert ist sind auch alle seine Threads terminiert.

3.5 Impementierung von Threads

- **User-Level Threads (ULT)**
  - Threads sind für den Kernel unsichtbar
  - Thread Management mittels Thread Library (Betriebssystem weiß nichts von Threads)
  - Thread Switching im User Mode → kein Mode Switch 2.10 notwendig
  - applikationsspezifisches Scheduling (da die Library auf alles mögliche angepasst werden kann)
  - **Threads Library**:

    - Enthält Code für:
        - Erzeugung von Threads
        - Terminierung von Threads
        - Daten- und Nachrichtenaustausch zwischen Threads
        - Thread Scheduling
        - Sichern und Herstellen von Thread Kontexten
    - Blockierender System Call blockiert **alle** ULTs eines Prozesses
    - keine Verteilung auf mehrere Prozessoren (da es für das Betriebssystem keine Threads gibt)

- **Kernel-Level Threads KLT**
    - Thread Management durch den Kernel
    - Kernel Thread API (keine Library)
    - Thread Switching durch den Kernel
    - Scheduling auf Thread-Basis
    - Thread-weises Blockieren (da der Kernel jeden Thread kennt)
    - Kernel-Routinen multi-threaded
    - gleichzeitiges Schedulen mehrerer Threads eines Prozesses (bei mehreren Prozessoren)
    - Thread Switching innerhalb eines Prozesses über den Kernel benötigt zwei Mode Switches 2.10 → langsamer als bei den ULTs
    - flexibleres Abarbeiten
    - Hybrid-Ansatz möglich mit ULTs innerhalb einer KLT Architektur

3.6 Zusammenfassung Threads

- **Prozess** als Einheit der Ressourcenverwaltung und **Thread** als Einheit des Dispatching.
- mehrere Threads pro Prozess möglich
- schnelleres Erzeugen und Umschalten als bei Prozessen
- verschiedene Implementierungen:
    - ULT 3.5
    - KLT 3.5

4 Mutual Exclusion und Synchronisation

Motivation ist ein kontrolliertes paralleles Laufen von Threads und Prozessen. Es soll durch eine von dem Entwickler spezifizierte Reihenfolge beibehalten werden.

4.1 Gemeinsame Ressourcen

Parallel laufende Prozesse verwenden die selben Daten und Ressourcen (Puffer, Shared Memory, Files, Geräte), tauschen Daten aus und / oder kooperieren. → es benötigt einen **disziplinierten Zugriff** auf diese gemeinsamen Ressourcen und eine **geordnete Abarbeitung** verteilter Aktionsfolgen. Wenn das nicht erfüllt ist kommt es zu Inkonsistenzen und nicht eindeutigen Ereignisfolgen.

4.2 Interaktion von Prozessen

- **Wettstreit um Ressourcen** (Competition)
- **Wechselseitiger Ausschluss** (Mutual Exclusion)
    - Verhindert das gleichzeitige Zugreifen auf Ressourcen
    - Ununterbrechbarkeit einer Aktionsfolge (Atomizität), „ganz oder gar nicht“
    - Ziel ist das Erhalten der Konsistenz der Daten

- **Bedingungssynchronisation** (Condition Synchronisation)
  - Erreichen einer definierten Abfolge von Operationen
  - Ziel ist das Vermeiden von Race Conditions (Abhängigkeit von Ergebnissen von relevantem Prozessfortschritt)

4.3 Mutual Exclusion und Conditional Synchronisation

Zwei unterschiedliche Formen der Synchronisierung, allerdings ist jede Kombination davon möglich:

MutEx.	*CondSync.*	*Ziel*
–	–	unabhängige Aktionen
–	+	vorgegebene Abfolge
+	–	Konsistenz
+	+	Konsistenz und Abfolge

Abbildung 16: Mutual Exclusion und Conditional Exclusion Kombinationen

4.3.1 Beispiele

- **Unabhängige Aktionen**: z.B. Fußgänger die denselben Gehsteig benutzen
- **vorgegebene Abfolge**: z.B. das Buffet sollte fertig sein, bevor die Gäste kommen, aber es ist dann egal in welcher Reihenfolge die Gäste essen usw.
- **Konsistenz**: z.B. in Wahlkabinen darf immer nur eine Person sein, allerdings ist das Ergebnis der Wahl unabhängig davon, in welcher Reihenfolge die Personen alleine in die Wahlkabine gegangen sind.
- **Konsistenz und Abfolge**: z.B. Verwendung von Shared Memory

4.4 Betriebssystem Kontrollaufgaben

Das Betriebssystem muss folgende Anforderungen garantieren:

- Mutual Exclusion
- Datenkonsistenz
- Regelung der Abfolgen der Ressourcenvergabe
- Verhinderung von einem Deadlock
- Verhinderung von Starvation (ein Prozess bekommt die CPU nicht mehr und wird *vergessen*)

4.5 Begriffe Mutual Exclusion

Der **kritische Abschnitt** ist teil des Prozesses, während die **kritische Region** der Teil im Speicher ist, wo es den Konflikt gibt. Ein Prozess befindet sich im **kritischen Abschnitt** wenn er auf gemeinsame Daten oder Ressourcen zugreift. → Lösung wäre Mutual Exclusion: Nur ein Prozess darf sich jeweils in einem kritischen Abschnitt befinden. Das bedeutet der Eintritt in einen kritischen Abschnitt muss geregelt erfolgen.

4.5.1 Prozessstruktur für kritischen Abschnitt

Kein Prozess darf in seinen kritischen Abschnitt eintreten, wenn sich bereits ein Prozess in seinem kritischen Abschnitt befindet.

Vor und nach dem kritischen Abschnitt muss es ein Protokoll geben. Der Epilog ist meist einfacher als der Prolog umzusetzen.

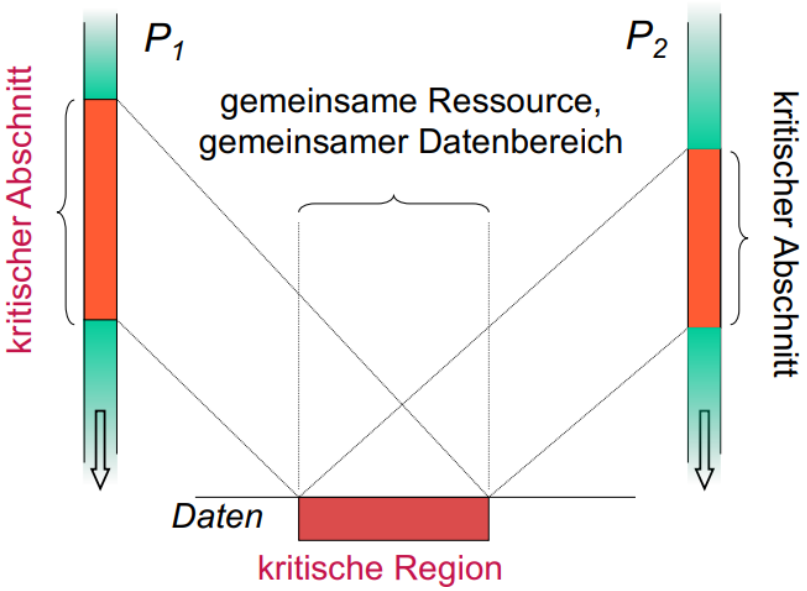

Abbildung 17: Mutual Exclusion Begriffe

Abbildung 18: Prozessstruktur für kritischen Abschnitt

4.5.2 Anforderungen für kritischen Abschnitt

- **Mutual Exclusion**: ist ja auch das was wir erreichen wollen
- **Progress**: wenn sich kein Prozess im kritischen Abschnitt befindet und Prozesse in den kritischen Abschnitt eintreten möchten, darf die Entscheidung über den nächsten Prozess nicht unendlich lange verzögert werden (kein Lifelock *„After You“*, im Sinne von zwei höflichen Menschen die sich gegenseitig den Vortritt geben). Durch zu langes Warten entsteht ein Lifelock → kein Prozess kommt voran, sperren sich aber nicht aus wie bei einem Deadlock (es könnte eigentlich weitergehen); **Lifelock = Aktivität ohne Produktivität**
- **Bounded Waiting**: Nachdem ein Prozess einen Request für den kritischen Abschnitt abgegeben hat, ist die Anzahl der Eintritte in den kritischen Abschnitt begrenzt. → Verhinderung einer Starvation. Kein Prozess sollte durch eine unfaire Warteschlange *verhungern*.

4.5.3 Lösungen

- **Softwarelösungen**: Annahme: Atomizität einer Lese- und Schreibeoperation auf dem Speicher; (Historische Lösung, es wird angenommen, es gebe nur Lese- und Schreibeoperationen, heutzutage haben mächtigere Instruktionen die auch atomar ausgeführt werden können)
- **Hardwareunterstützte Lösungen**: mächtigere atomare Maschineninstruktionen
- **Höhere Synchronisationskonstrukte**: stellen dem Programmierer Funktionen und Datenstrukturen zur Verfügung: **Semaphore, Monitor, Massage Passing, ...**

4.6 Mutual Exclusion in Software

Annahmen:

- 2 Prozesse P_0 und P_1
- dann Generalisierung auf n Prozesse $\rightarrow P_i$ wobei $P_i \neq P_j$ für $i \neq j$
- Wir nehmen an dass auf der Hardware nur atomares Lesen und Schreiben eines Wertes möglich ist
- Die Synchronisierung erfolgt über globale Variablen
- Busy Waiting: Warten auf das Eintreten einer Bedingung durch ständiges Testen

4.6.1 Dekker-Algorithmus

```
flag[i] := true;
while flag[j] do
    if turn = j then
    begin
        flag[i] := false;
        while turn = j do nothing;
        flag[i] := true;
    end;
// critical section
turn := j;
flag[i] := false;
// remainder section
```

`flag[]` beschreibt den Zustand der Prozesse; `turn` bestimmt die Reihenfolge des Eintretens in den kritischen Abschnitt im Falle von völligem Gleichlauf (beide Prozesse würde zufällig exakt parallel ablaufen)

4.6.2 Peterson-Algorithmus

```
loop
    flag[i] := true;
    turn := j;
    while flag[j] and turn = j do nothing;
    // critical section
    flag[i] := false;
    // remainder section
end loop;
```

Als Initialisierung wird jeder wert im `flag` Array auf `false` gesetzt. (Dieser Algorithmus ist eigentlich dasselbe wie der Dekker-Algorithmus nur leichter zu lesen.)

4.6.3 Bakery-Algorithmus

Lösung für das kritische-Abschnitt-Problem für n Prozesse

- Nummernvergabe für kritische Abschnitt; Prozesse mit der niedrigsten Nummer (> 0) tritt als erstes ein
- Nummer 0 heißt „keine Anforderung für kritischen Abschnitt“
- P_i und P_j mit gleichen Nummern: P_i kommt vor P_j in den kritischen Abschnitt wenn $j < j$ (der Prozess mit der kleineren Prozessnummer kommt zuerst dran)
- es gilt immer: neu vergebene Nummer $\geq$ vorher vergebene Nummern

```
// initialization

var choosing: array[0 .. n - 1] of boolean;
var number: array[0 .. n - 1] of integer;
```

```
loop
    // choose or compute a number
    choosing[i] := true;
    number[i] := 1 + max(number[0], ..., number[n - 1]);
    choosing[i] := false;

    // for every process
    for j := 0 to n - 1 do
    begin
        // wait for j to finish choosing a number
        while choosing[j] do nothing;

        // wait until j does not need to go into the critical section
        // and the number of j is larger
        while number[j] != 0 and (number[j], j) < (number[i], i) do nothing;
    end;
    // critical section
    number[i] := 0;
    // remainder section
end loop;
```

4.7 Hardwareunterstütze Lösungen

- **Interrupt Disabling während kritischem Abschnitt**:
  - für Uniprozessoresysteme geeignet
  - Einschränkung des Betriebssystem beim Dispatching anderer Tasks
  - Eingriff in die Autonomie des Betriebssystems
  - Was passiert bei einem Fehler in einem kritischen Abschnitt? (Endless Loop)
  - Rückgabe der Kontrolle an das Betriebssystem?
  - Schutz von Instruktionssequenzen im Betriebssystem
- **Mächtigere atomare Maschineninstruktionen**:
  - Test and Set 4.7.1 (Teste die Variable und setzte sie bedingt); wird auf fast jedem Microprozessor gefunden
  - Exchange 4.7.2 (= swap)

4.7.1 Test and Set

Hardwareinstruktionen die zwei Aktionen atomar ausführt → Mutual Exclusion

```
function testset(var i: integer): boolean;
begin
    // start of the atomic region
    if i = 0 then
    begin
        i := 1;
        return true;
    end;
    else
    begin
        return false;
    end;
    // end of the atomic region
end;
```

Das ist in Wirklichkeit eine Maschineninstruktion, allerdings leichter lesbar im Pseudocode. Diese Instruktion kann so verwendet werden:

```
var b := 0 : integer;
```

```
while not testset(b) do nothing;
// critical section
b := 0;
// remainder section
```

Das Überprüfen und setzten der `flag` Variable kann dadurch atomar ausgeführt werden. → nur ein Prozess kann in den kritischen Abschnitt eintreten.

4.7.2 Exchange

Maschineninstruktion die die Werte zweier Variablen vertauscht.

```
var b := 0 : integer;
```

```
var key := 0 : integer;

key := 1;
do exchange(key, b) while key = 1;
// critical section
exchange(key, b);
// remainder section
```

Einfache Prolog für den kritischen Abschnitt für beliebig viele Prozesse, allerdings ist Starvation möglich.

4.8 Busy Waiting

Alle bisherigen Lösungen haben Busy Waiting verwendet, das bedeutet sie iterieren in einer Schleife bis eine bestimmte Synchronisationsbedingung erfüllt ist → verschwenden CPU Zeit mit warten. Außerdem ist das **kein geeigneter Mechanismus** zur Synchronisation für Benutzer-Code → **Blockieren der wartenden Prozesse** in Blocked Queues **als Lösung** zu Busy Waiting.
Busy Waiting ist **keine** Synchronisationsmethode! (Ausnahme in Low-Level Bereichen ohne Betriebssysteme).

»Es ist ein Verbrechen CPU Zeit auf diese Art und Weise so zu verbraten. Haben wir uns Verstanden?«

4.9 Semaphore

Eine Semaphore (eigentlich Bahnsignal) ist eine Technik um Synchronisieren ohne Busy Waiting zu ermöglichen.

- Synchronisationskontrukt ohne Busy Waiting (vom Betriebssystem zur Verfügung gestellt)
- Semaphor S: Integer-Variable, auf die nach der Initialisierung nur mit zwei atomaren Funktionen, `wait()` und `signal()`, zugegriffen werden kann. Die Funktionen inkrementieren bzw. dekrementieren den Wert des Semaphores.

Wunschverwendung: Sichern eines kritischen Abschnitts auf eine *einfache Art*:

```
wait(S);
// critical section
signal(S);
// remainder section
```

4.9.1 Semaphore: Datenstruktur

Ein Semaphor ist in Wirklichkeit nicht nur eine Integer Variable sondern inkludiert auch noch eine Queue.

```
type semaphore = record
    value: integer;
    queue: list of processes;
end;
```

Auf einen Semaphor S wartende Prozesse werden in die Blocked Queue von S *geparkt*.

```
init(S, val):
    S.value := val;
    S.queue := empty list;
```

```
wait(S):
    S.value := S.value - 1;
    if S.value < 0 then
        add this process to S.queue and block it;
```

```
signal(S):
    S.value := S.value + 1;
    if S.value <= 0 then
        remove a process P from S.queue and place P on ready queue;
```

Semaphoren müssen auf einen nicht-negativen Wert (für Mutual Exclusion auf 1) initialisiert werden.

4.9.2 Semaphore: Mutual Exclusion

```
init(S, 1);
```

```
wait(S);
// critical section
signal(S);
// remainder section
```

- Maximal ein Prozess im kritischen Abschnitt
- Prozesse treten im FIFO Modell (Queue) ein; Bounded Waiting 4.5.2, Fairness

4.9.3 Semaphore: Implementierung

- `wait()` und `signal()` müssen als atomare Operationen ausgeführt werden
- Sichern der Semaphorenoperationen mit Test and Set 4.7.1 (sind auch kritische Abschnitte)
  - zusätzliche Record-Komponente `flag` in der Semaphorendatenstruktur
  - vor kritischem Abschnitt Busy Waiting mit Test von `flag`
  - da `wait()` und `signal()` sehr kurz sind, kommt es kaum zum Busy Waiting → Overhead vertretbar

4.9.4 Semaphore: Bemerkungen

- `S.count` $\geq$ `0`: Anzahl der Prozesse, die hintereinander ein `wait()` ausführen können, ohne zu blockieren
- Initialisierung von S mit Wert $n \rightarrow n$ Prozesse können gleichzeitig auf die Ressource zugreifen (allgemeinere Synchronisationsaufgaben z.B. mit n Ressourcen)
- `S.count < 0`: |`S.count`| Prozesse in der Queue von S blockiert (in der gezeigten Implementierung)
- **Binary Sempahore**: nimmt nur die Werte 0 oder 1 an
- **Counting Semaphore**: kann beliebige (ggf. nicht-negative) ganzzahlige Werte annehmen (implementierungsabhängig)
- Operationen `wait()` und `signal()` werden in der Literatur auch als `P` bzw. `V` bezeichnet.

4.9.5 Bedingungssynchronisation

In diesem Fall gibt es zwei Prozesse P_1 und P_2 wobei der Codeabschnitt C_1 in P_1 vor Abschnitt C_2 ausgeführt werden soll. → Semaphor als Bedingungssynchronisation.

```
// initialization of S

init(S, 0);
```

```
// P1

// C1
signal(S);

```

```
// P2

wait(S);
// C2

```

4.9.6 Abwechselnder Zugriff

In diesem Beispiel geht es um einen Datentransfer von P_1 zu P_2 ohne dass es ein Duplizieren oder Verlust der Daten gibt. (P_1 liest, P_2 schreibt)

```
// initialization of S1 and S2

init(S1, 1);
init(S2, 0);
```

```
// P1

loop
    generate data;
    wait(S1);
    write ShM;
    signal(S2);
end loop;

```

```
// P2

loop
    wait(S2);
    read ShM;
    signal(S1);
    use data;
end loop;

```

4.9.7 Aufgabe

Aufgabenstellung

- Gegeben sind 3 zyklische Prozesse A, B, C
- A produziert Daten und schreibt sie auf ShM
- B und C lesen die Daten vom ShM
- Jeder von A geschriebene Datensatz soll von B und C genau einmal gelesen werden.

Lösung

```
// initialization of the semaphores

init(S1, 2);
init(S2, 0);
init(S3, 0);
```

```
// A

loop
    generate data;
    wait(S1);
    wait(S1);
    write ShM;
    signal(S2);
    signal(S3);
end loop;

```

```
// B

loop
    wait(S2);
    read ShM;
    signal(S1);
    use data;
end loop;

```

```
// C

loop
    wait(S3);
    read ShM;
    signal(S1);
    use data;
end loop;

```

4.9.8 Beispiele

Producer-Consumer Problem

- Produzenten generieren Daten, Konsumenten lesen diese ein (z.B. Druckaufträge)
- einzelne Datensätze werden über einen Puffer an Konsumenten übergeben

Producer-Consumer mit unbegrenztem Puffer

- Produzent kann jederzeit ein Datenelement schreiben
- Konsument muss auf Daten warten
- `in` zeigt auf nächstes freies Puffer-Element
- `out` zeigt auf nächstes zu lesendes Element

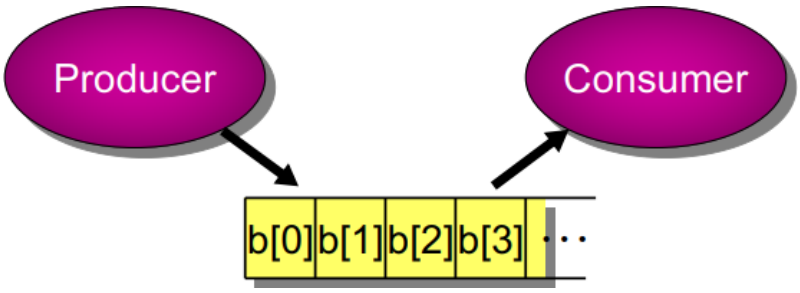

Abbildung 19: PC Problem

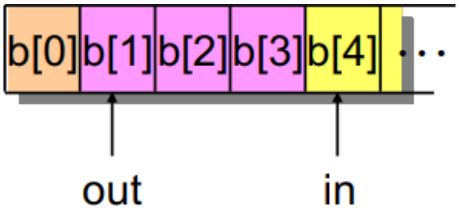

Abbildung 20: PC Problem Puffer Struktur

Producer-Consumer: Probleme

- Mutual Exclusion: zu jedem Zeitpunkt darf nur ein Prozess auf den Puffer zugreifen (Semaphore S)
- Bedingungssynchronisation: ein Konsument darf nur dann lesen wenn mindestens ein ungelesenes Datenelement im Puffer steht (Counting Semaphore N spiegelt die Anzahl der Elemente im Puffer wieder)

Producer-Consumer: Implementierung

```
// initialization

init(S, 1);
init(N, 0);
in := 0 : integer;
out := 0: integer;
```

```
// program

function append(v):
    b[in] := v;
    in := in + 1;

function take():
    w := b[out];
    out := out + 1;
    return w;
```

```
// producer

loop
    produce(v);
    wait(S);
    append(v); // critical section
    signal(S);
    signal(N);
end loop;

```

```
// consumer

loop
    wait(N);
    wait(S);
    w := take(); // critical section
    signal(S);
    consume(w);
end loop;

```

Producer-Consumer mit Ringpuffer

- Begrenzter Puffer mit k Elementen
- Lesen: mindestens ein neuer Wert notwendig
- Schreiben: mindestens ein Element frei

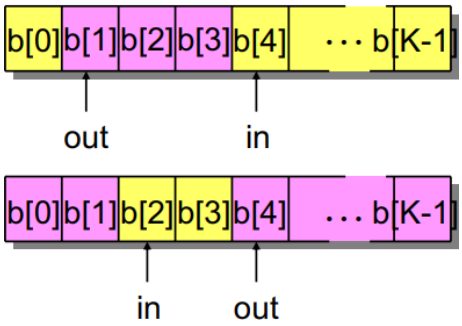

Abbildung 21: PC Problem Ringpuffer Struktur

Producer-Consumer Ringpuffer: Implementierung

```
// initialization

init(S, 1);
init(N, 0);
init(E, k);
in := 0 : integer;
out := 0 : integer;
```

```
// program

function append(v):
    b[in] := v;
    in := (in + 1) mod k;

function take():
    w := b[out];
    out := (out + 1) mod k;
    return w;
```

```
// producer

loop
    produce(v);
    wait(E);
    wait(S);
    append(v); // critical section
    signal(S);
    signal(N);
end loop;
```

```
// consumer

loop
    wait(N);
    wait(S);
    w := take(); // critical section
    signal(S);
    signal(E);
    consume(w);
end loop;
```

In diesem Fall ist die Reihenfolge von den `signal()` Funktionsaufrufen egal, aber die Reihenfolge von `wait()` wichtig!
Faustregel: Alle `wait()` Aufrufe für einen kritischen Abschnitt so kurz wie möglich direkt vor dem kritischen Abschnitt platzieren.

Reader-Writer Problem

- Es gibt Lese- und Schreibeprozesse, die auf eine gemeinsame Ressource zugreifen.
- beliebig viele Leser dürfen parallel lesen (Parallelität wird erwünscht)

- Schreiben benötigen exklusiven Zugriff

Ein Beispiel wäre der Lehrer, der an die Tafel schreibt, und die Schüler, die alle gleichzeitig davon lesen.

Reader-Writer: Implementierung

```
// initialization

init(X, 1);
init(Y, 1);
init(Z, 1);
init(Wsem, 1);
init(Rsem, 1);
rc := 0 : integer;
wc := 0 : integer;
```

```
// reader

loop
    wait(X);
    rc := rc + 1;
    if rc = 1 then wait(Wsem);
    signal(X);
    read; // critical section
    wait(X);
    rc := rc - 1;
    if rc = 0 then signal(Wsem);
    signal(X);
end loop;

```

```
// writer

loop
    wait(Wsem);
    write; // critical section
    signal(Wsem);
end loop;

```

In diesem Fall haben die Reader Prioriät.

```
// reader

loop
    wait(Z);
    wait(Rsem);
    wait(X);
    rc := rc + 1;
    if rc = 1 then wait(Wsem);
    signal(X);
    signal(Rsem);
    signal(Z);
    read; // critical section
    wait(X);
    rc := rc - 1;
    if rc = 0 then signal(Wsem);
    signal(X);
end loop;

```

```
// writer

loop
    wait(Y);
    wc := wc + 1;
    if wc = 1 then wait(Rsem);
    signal(Y);
    wait(Wsem);
    write; // critical section
    signal(Wsem);
    wait(Y);
    wc := wc - 1;
    if wc = 0 then signal(Rsem);
    signal(Y);
end loop;

```

In diesem Fall haben die Writer Priorität.

```
// reader

loop
    wait(Z);
    wait(Rsem);
    wait(X);
    rc := rc + 1;
    if rc = 1 then wait(Wsem);
    signal(X);
    signal(Rsem);
    signal(Z);
    read; // critical section
    wait(X);
    rc := rc - 1;
    if rc = 0 then signal(Wsem);
    signal(X);
end loop;

```

```
// writer

loop
    wait(Rsem);
    wait(Wsem);
    write; // critical section
    signal(Wsem);
    signal(Rsem);
end loop;

```

In diesem Fall haben die Writer die Priorität und es gibt keine Starvation der Reader.

4.9.9 Semaphore: Probleme

- Semaphorenoperationen sind über Prozesse verteilt → schwer zu debuggen und analysieren
- Operationen müssen in allen Prozessen korrekt verwendet werden
- Ein fehlerhafter Prozess bewirkt Fehlverhalten aller Prozesse, die zusammenarbeiten

Lösung → Monitor (nicht dasselbe wie Monitor aus 1960 1.1.2)

4.10 Monitor, Nachrichten und andere Synchronisationsmechanismen

4.10.1 Monitor

- **Softwaremodul**, bestehend aus:
  - Prozeduren
  - lokalen Daten
  - Initialisierungscode
- **Eigenschaften**
  - Zugriff auf lokale Variable: Monitorprozeduren
  - Eintritt von Prozessen in den Monitor über Monitorprozeduren
  - max. 1 Prozess zu jedem Zeitpunkt im Monitor

Monitor: Eigenschaften

- Monitor sorgt für Mutial Exclusion, kein explizites Programmieren
- Gemeinsamer Speicher ist im Monitorbereich anzulegen
- Bedingungssynchronisation über Monitorvariable
  - Programmieren von Wartebedingungen für Bedingungsvariable (Conditional Variables)
  - Monitoreintritt, wenn Wartebedingungen der Bedingungsvariablen erfüllt ist

Monitor: Conditional Variables

- Lokal im Monitor (Zugriff nur im Monitor)
- Zugriff nur über die Zugriffsfunktionen
  - `cwait(c)`: blockiert aufrufenden Prozess bis Bedingung(svariable) `c` den Wert `true` annimmt
  - `csignal(c)`: Setze einen Prozess, der auf die Bedingung `c` wartet fort
    - wenn mehrere Prozesse warten → wähle einen aus
    - wenn kein Prozess wartet → keine Aktion
    - keine speichernde Wirkung (nicht wie beim Semaphor-`wait()`)

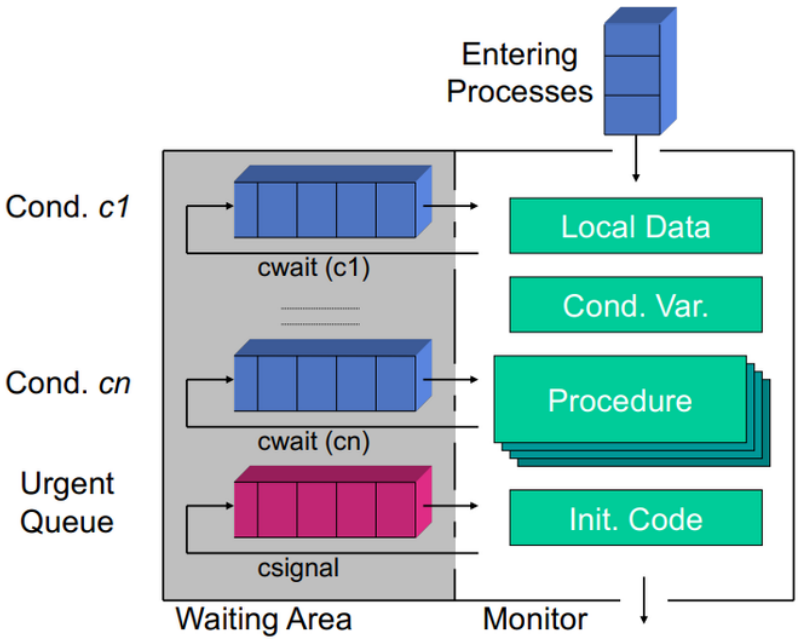

Abbildung 22: Monitor Diagramm

Monitor: Implementierung Diese Implementierung zeigt das Producer-Consumer mit dem Ringpuffer Problem.

```
// inside of the monitor
// initialization

b: array[0 .. k - 1] of items;
in := 0 : integer;
out := 0 : integer;
cnt := 0 : integer; // count of items in the ringbuffer
notfull : condition;
notempty : condition;
```

```
// inside of the monitor
// monitor procedure

function append(v):
    if cnt = k then cwait(notfull);
    b[in] := v;
    in := (in + 1) mod k;
    cnt := cnt + 1;
    csignal(notempty);

function take(v):
    if cnt = 0 then cwait(notempty);
    v := b[out];
    out := (out + 1) mod k;
    cnt := cnt - 1;
    csignal(notfull);
```

4.10.2 Message Passing

- Methode zur Interprozesskommunikation (IPC), für einzelne Computer und verteilte Systeme
- Eine Nachricht ist eine **atomare Datenstruktur** → Datenkonsistenz
- verschiedene Synchronisationssemantiken
- Funktionen zum Senden und Empfangen (atomare Funktionen)
  - `send(destination, message)`
  - `receive(source, message)`

Charakteristika von Nachrichten

- **Synchronisation** (Verschiedene Arten wie sich `send()` und `receive()` verhalten)
  - `send()`
    - **blocking**: `send()` blockiert bis alle Empfänger die Nachricht erhalten haben
    - **non-blocking**: `send()` sendet Daten aus und warten nicht auf die Empfänger
  - `receive()`
    - **blocking**: `receive()` blockiert so lange bis eine Nachricht erhalten wurde
    - **non-blocking**: `receive()` wird ausgeführt, wenn gerade eine Nachricht eingelangt wird diese ausgewertet ansonsten passiert nichts; meist anderer return-code
    - **test for arrival**: abgespeckte Version von `receive()` gibt nur zurück ob Nachrichten da sind
- **Interpretation**
  - **Ereignisnachrichten**: berichtet über das Eintreten eines Ereignisses (z.B. neuer Datensatz verfügbar); Normalerweise müssen solche Nachrichten *konsumiert* werden. Verwendung beim Producer Consumer Problem 4.9.8.
  - **Zustandsnachrichten**: berichten über den Aktuellen Zustand z.B. über einer Variable; Solche Nachrichten müssen nicht *konsumiert* werden, können aber auch öfters ausgelesen werden. Beispielsweise kann eine Zustandsnachricht über den Zustand einer Variable jede Minute gesendet werden, der Empfänger kann sich die aktuelle Nachricht dann ansehen wenn er über den Zustand Bescheid wissen will. Verwendung beim Reader Writer Problem 4.9.8.
- **Datenverwaltung**
  - **Ereignissemantik**: Warteschlange (Queue); Wenn z.B. mehrere Datensätze verfügbar sind, so müssen diese in einer Queue abgespeichert werden, um dann nach und nach von dem Empfänger oder den Empfängern abgearbeitet zu werden.
  - **Zustandssemantik**: empfangene Daten überschreiben alte Werte, um den aktuellen Zustand einsehen zu können
- **Adressierung**
  - **1 : 1 vs. 1 : N**: einzelne Empfänger vs. eine Gruppe von Empfängern
  - **physikalisch vs. logisch**: an Treitlstraße 3 vs. „Hol mir einen Arzt“
  - **direkt vs. indirekt**: direkt an einen Prozess vs. Mailbox, Portadressierung 23

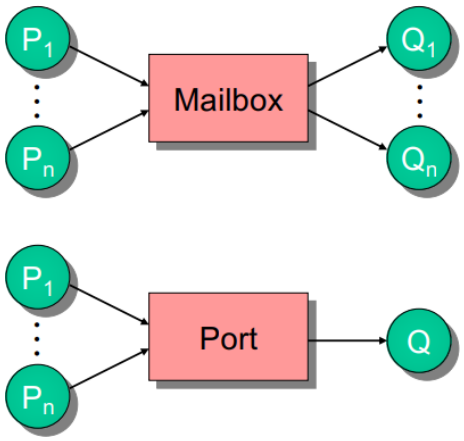

Abbildung 23: Mailbox und Portadressierung

Bei der Mailbox gibt es mehrere Prozess, die die Nachrichten abholen, bei der Portadressierung nur einen. Über eine Mailbox kann auch Mutual Exclusion realisiert werden.

Ereignisnachricht für Mutual Exclusion

- Prozesse verwenden eine gemeinsame Mailbox *mutex* und eine Nachricht *token*
- `receive()`: blocking
- `send()`: non-blocking

```
// initialization

send(mutex, msg);
```

```
// in process P_i

loop
    receive(mutex, msg);
    // critical section
    send(mutex, msg);
    // remainder section
end loop;
```

Der Prozess der den token hat, darf auf in den kritischen Abschnitt eintreten. Das `receive()` blockiert so lange bis dieser Prozess den Token erhält, kann dann in den kritischen Abschnitt eintreten und danach den Token wieder in die Mailbox abgeben.

4.10.3 Locks

- Einfacher Synchronisationsmechanismus
- zwei Funktionen:
  - `enter(lock)`: blockiert nachfolgende `enter()`-Aufrufe
  - `release(lock)`: gibt das Lock frei

4.10.4 Sequencer und Eventcounter

- **Eventcounter** E: ganzzahliger Ereigniszähler
  - `advance(E)`: erhöht E um 1 (Anfangswert: 0)
  - `await(E, val)`: blockiert bist $E \geq$ `val`
- **Sequencer** S: ganzzahliger Nummernserver
  - `ticket(S)`: retourniert Wert von S und inkrementiert S um 1. (Anfangswert: 0)

```
myTicket = ticket(S);
await(E, myTicket);
// critical section
advance(E);
```

4.10.5 Zusammenfassung

- Anforderungen paralleler Prozesse
  - Konsistenter Zugriff auf gemeinsame Daten (Mutual Exclusion)
  - Vorgegebene Reihenfolge von Aktionen (Conditional Synchronisation)
- kritischer Abschnitt
  - Aktionen, die gemeinsame Daten manipulieren
  - mit Synchronisationskonstrukten gesichert
- Sicherung des kritischen Abschnitts
  - Lösungen abhängig vom Instuction Set
  - Unterstützung durch Betriebssystem

  - häufige auftretende Probleme → mächtigere Instruktionen anstelle von schwierigen Algorithmen
  - kein Busy Waiting sondern blocked Zustand, der vom Betriebssystem zur Verfügung gestellt wird
- Semaphore
  - `init()`, `wait()` bzw. `P()` und `signal()` bzw. `V()`
  - Reihenfolge der `wait()` Operationen wichtig!
- Monitor
- Nachrichten

5 Deadlock

- Permanentes Blockieren einer Menge von Prozessen, die um Ressourcen konkurrieren oder miteinander kommunizieren
- **Zyklischer Ressourcenkonflikt** zwischen zwei oder mehreren Prozessen:
  1. jeder Prozess hält eine Ressource und
  2. wartet auf eine Ressource, die gerade ein anderer Prozess hält
- erneuerbare vs. konsumierbare Ressourcen
  - **erneuerbar**: z.B. allokierter Speicher im Arbeitsspeicher; kann wieder freigegeben werden → Prozess der blockiert, weil nicht genügend Speicher frei ist, kann später wieder anlaufen.
  - **konsumierbar**: z.B. Nachrichten; Wenn man auf eine Nachricht wartet, die schon konsumiert wurde, wartet man unendlich lange
- es gibt keine universelle Lösung des Problems → Überblick und leichte Verständlichkeit wichtig bei Synchronisationskonstrukten
- es gibt keine universelle Möglichkeit zu Prüfen ob ein Deadlock entstehen kann

5.1 Beispiele

5.1.1 Nachrichten

Synchronisation über Nachrichten (konsumierbare Ressourcen). In diesem Fall sind das blockierende `receive()` Aufrufe.

```
// P1

// ...
receive(MQ1, msg);
// ...
send(MQ2, msg);
// ...

```

```
// P2

// ...
receive(MQ2, msg);
// ...
send(MQ1, msg);
// ...

```

5.1.2 Mututal Exclusion

Prozesse greifen auf zwei gemeinsame Ressourcen zu

```
// P1

// ...
Get B
// ...
Get A
// ...
Release B
// ...
Release A
// ...

```

```
// P2

// ...
Get A
// ...
Get B
// ...
Release B
// ...
Release A
// ...

```

```
// P3

// ...
Get A
// ...
Release A
// ...
Get B
// ...
Release B
// ...

```

5.2 Prozessfortschrittsdiagramm

Ereignisfolgen im Bezug auf die Ressourcenverwendung. Jede grüne Linie stellt einen möglichen Trace, ein mögliche Abarbeitung der Prozesse, dar. Ein Prozessfortschrittsdiagramm kann nur in einfachen Fällen angefertigt werden (z.B. wenn keine Schleifen vorhanden sind) und kann analytisch auch nicht gut ausgewertet werden → keine Lösung um herauszufinden, ob ein Deadlock entstehen kann.

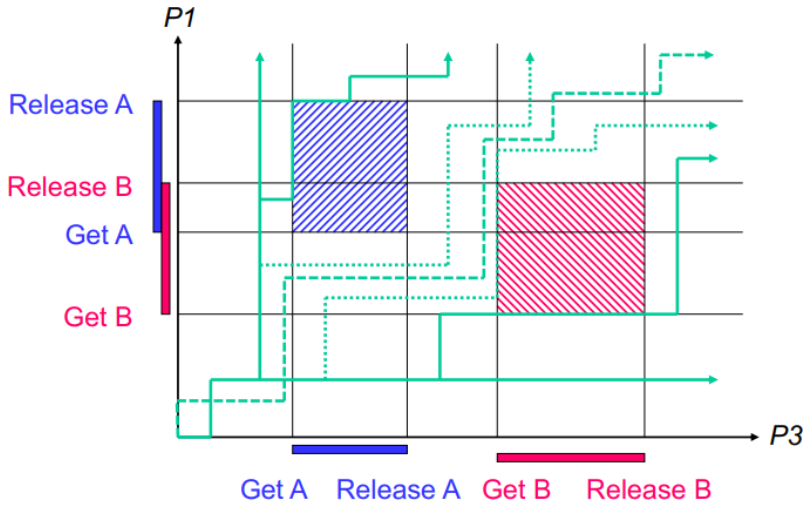

Abbildung 24: Prozessfortschrittsdiagramm zwischen P_1 und P_3

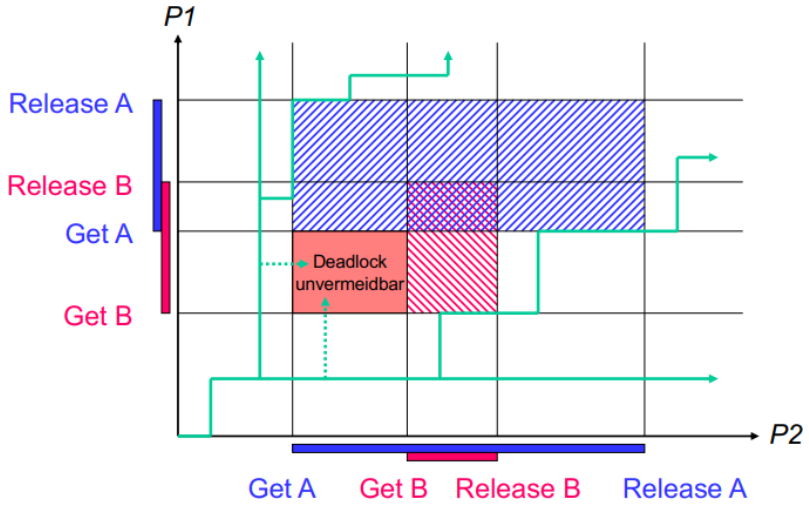

Abbildung 25: Prozessfortschrittsdiagramm zwischen P_1 und P_2

5.3 4 Deadlock-Bedingungen

- Ein Deadlock tritt genau dann auf, wenn **Circular Wait** nicht aufgelöst werden kann
- **Circular Wait** kann nicht aufgelöst werden, wenn die Bedingungen 1 bis 3 gelten

Daraus folgt: **Bedingungen 1 bis 4 sind notwendig und hinreichend für das Auftreten eines Deadlocks.**

5.3.1 3 Voraussetzungen des Systems

1. **Mutual Exclusion**: exklusiver Zugriff auf Ressourcen
2. **Hold and Wait**: Prozess kann Ressourcen halten, während er auf andere Ressourcen wartet
3. **No Preemption**: zugewiesene Ressourcen werden den Prozessen nicht weggenommen

5.3.2 1 bestimmte Ereignisfolge

4. **Circular Wait**: Geschlossene Kette von Prozessen, von denen jeder Prozess mindestens eine Ressource hält, die von einem anderen Prozess benötigt wird.

5.4 Behandlung von Deadlocks

- **Deadlock Prevention**: verhindern einer der 4 Deadlock-Bedingungen
- **Deadlock Avoidance**: Ressourcenreservierungen, die zu Deadlocks führen könnten, werden nicht gewährt
- **Deadlock Detection**: Ressourcenanforderungen werden immer gewährt, sofern Ressourcen vorhanden sind plus **periodisches Überprüfen**, ob ein Deadlock vorliegt mit ggf. **Recovery**

5.5 Deadlock Prevention

- Betriebssystemdesign, das Deadlock ausschließt
- **Indirekt Deadlock Prevention**: Verhinderung einer Bedingung 1 bis 3
- **Direkt Deadlock Prevention**: Verhinderung der Bedingung 4 (Circular Wait)

5.5.1 Indirect Deadlock Prevention

- **Mutual Exclusion**: Kann nicht unterbunden werden, ist ein essenzieller Bestandteil der Parallelität
- **Hold and Wait**:
  - Eine Lösung wäre eine kleine Abänderung, sodass der Process alle Ressourcen am Start anfordert und am Ende alle auf einmal frei gibt. (analog zum konservativen 2PL)
    1. Prozesse fordern alle Ressourcen auf einmal an
    2. Blockieren, bis alle Ressourcen vorhanden sind
  - **Nachteile**:
    * Lange Verzögerung der Prozesse möglich
    * schlechte Nutzung allokierter Ressourcen
    * Prozess braucht Wissen über Ressourcen, die er verwenden wird
- **No Preemption**:
  - Anwendbar für Ressourcen deren Zustand leicht gespeichert und später wieder hergestellt werden kann (z.B. Prozessor, beim Blockieren von `wait()`)
    1. Prozess gibt Ressourcen frei, wenn er eine weitere Ressource nicht bekommt
    2. Anforderung eines Prozesses führt dazu, dass ein anderer Prozess Ressourcen freigeben muss

5.5.2 Direct Deadlock Prevention

- **Protokoll: Verhinderung des Circular Wait**
  - Strikte lineare Ordnung O für Ressourcenarten; z.B.:

$$O(\text{Tape Drives}) = 2$$
$$O(\text{Disk Drives}) = 4$$

  - Erste Anforderung: Prozess fordert eine Anzahl von Ressourcen der Art R_i **auf einmal** an
  - in Folge werden für den Prozess nur Anforderungen von Ressourcen der Art R_k mit $O(R_k) > O(R_i)$ zugelassen
  - **Nachteil**: Ineffizient durch unnötiges Zurückweisen von Anforderung aufgrund der Ordnung

5.6 Deadlock Avoidance

- Bedingungen 1 bis 3 erlaubt
- selektives Vergeben von Ressourcen
  - **Process Initiation Denial**: ein Prozess wird nicht gestartet, wenn seine Anforderungen zu einem Deadlock führen können; Es wird überprüft ob die Ressourcenanforderung eines Prozesses in Kombination mit den bereits laufenden Prozessen zu einem Deadlock führen kann, wenn ja → Prozess wird gar nicht erst gestartet
  - **Ressource Allocation Denial**: eine Ressourcenanforderung eines Prozesses wird verwehrt, wenn dadurch ein Deadlock entstehen könnte; Wenn ein Prozess neue Ressourcen anfordert wird überprüft ob es zu einem Deadlock führen kann, wenn ja → Prozess wird verzögert
- führt zu höherer Parallelität als beim Deadlock Prevention
- **Nachteil**: Ressourcenbedarf der Prozesse muss im Vorhinein bekannt sein

5.6.1 Deadlock Avoidance: Notation

- n Prozesse
- m Ressourcenkategorien
- Vektoren
  - Gesamtzahl der Ressourcen im System; Wie viele Ressourcen es aus jeder Kategorie gibt

$$\text{Resource} = (R_1, R_2, ..., R_m)$$

  - Anzahl der nicht vergebenen Ressourcen; Wie viele Ressourcen aus jeder Kategorie frei sind

$$\text{Available} = (V_1, V_2, ..., V_m)$$

- Matrizen
  - Maximalanforderungen an Ressourcen; Beschreibt Prozess in seiner gesamten Lebensdauer

$$\text{Claim} = \begin{pmatrix} C_{11} & C_{12} & ... & C_{1m} \\ ... & ... & ... & ... \\ C_{n1} & C_{n2} & ... & C_{nm} \end{pmatrix}$$

  - gegenwärtige Ressourcenbelegung

$$\text{Allocation} = \begin{pmatrix} A_{11} & A_{12} & ... & A_{1m} \\ ... & ... & ... & ... \\ A_{n1} & A_{n2} & ... & A_{nm} \end{pmatrix}$$

5.6.2 Process Initiation Denial

Ein Prozess P_{n+1} wird nur gestartet, wenn seine Ressourcenanforderungen keinen Deadlock hervorrufen können, d.h. wenn:

$$R_i \geq C_{(n+1)i} + \sum^{n} k = 1$$

Nachteil ist die Annahme, dass alle Prozesse ihre Ressourcen gleichzeitig anfordern → Einschränkung der Parallelität

5.6.3 Resource Allocation Denial: Banker's Algorithm

Ein **State** ist die Ressourcenbelegung der Prozesse, charakterisiert durch Vektoren und Matrizen; Kommt zum Einsatz wenn ein bereits laufender Prozess Ressourcen dazu allokieren möchte

- **Safe State**: es existiert eine Folge von Schritten, mit der der Algorithmus alle Prozesse als *finished* markiert in den Endzustand bringt
- **Unsafe State**: es existiert keine solche Schrittfolge

5.6.4 Banker's Algorithm: Notation

Zusätzliche Matrix:

$$\text{Required} = \begin{pmatrix} N_{11} & N_{12} & ... & N_{1m} \\ ... & ... & ... & ... \\ N_{n1} & N_{n2} & ... & N_{nm} \end{pmatrix}$$

Wobei gilt:

$$N_{ki} = C_{ki} - A_{ki}$$

Vor einer Ressourcenzuweisung wird ein Test durchgeführt, ob die Zuweisung zu einem Safe State führt:

- Safe State: Zuteilung der Ressourcen
- Unsafe State: keine Zuteilung der Ressourcen

5.6.5 Banker's Algorithm: Implementierung

```
// initialization

markiere alle Prozesse als unfinished
Work Vector W_i = V_i fuer alle i
```

```
loop
    suche P_k mit unfinished(P_k) und N_ki ≤ W_i fuer alle i
    kein Prozess gefunden → goto END
    andernfalls → markiere Prozess als finished und gib seine Ressourcen frei:
    W_i = W_i + A_ki fuer alle i
end loop;

END: alle Prozesse mit finished markiert → Safe State
andernfalls → Unsafe State
```

5.6.6 Banker's Algorithm: Anwendung

Test, ob aktuelle Ressourcenanforderungen Q_{ki} von Prozess P_k erfüllt werden sollen

1. Test: $Q_{ki} \leq N_{ki}$ für alle i
   - `true` → weiter
   - `false` → Fehler (Anforderungen zu hoch)
2. Test: $Q_{ki} \leq V_i$ für alle i
   - `true` → weiter
   - `false` → Warte (Ressourcen nicht verfügbar)
3. Provisorisches Gewähren der Anforderungen und Untersuchen auf Safe State

   für alle i :

   $$V_i := V_i - Q_{ki}$$
   $$A_{ki} := A_{ki} + Q_{ki}$$
   $$N_{ki} := N_{ki} - Q_{ki}$$
4. Test: Ist der vorliegende State ein Safe State?
   - `true` → weise Ressourcen Q_{ki} an P_k zu
   - `false` → Aufschieben des Requests

5.6.7 Banker's Algorithm: Bemerkung

- Safe State → kein Deadlock möglich
- Unsafe State → Deadlock möglich, muss aber nicht eintreten
  - Prozesse benötigen Ressourcen nicht während der gesamten Ausführungsdauer
  - Maximalanforderungen in den Ressourcenkategorien treten nicht gemeinsam auf
- Strategien zur Deadlock Avoidance nehmen Unabhängigkeit der Prozesse an (keine Bedingungssynchronisation zwischen Prozessen)

5.7 Deadlock Detection

- Ressourcenanforderungen werden immer gewährt, sofern Ressourcen vorhanden sind
- Notwendige Betriebssystem-Vorkehrungen
  - Algorithmus, um Deadlocks zu erkennen
  - Strategie zur Deadlockbehebung (Recovery)
- Deadlocküberprüfung z.B. bei jeder Ressourcenanforderung → CPU-intensiv

5.7.1 Deadlock Detection: Algorithmus

```
// initialization

setze alle Prozesse P_k auf unmarkiert
```

```
markiere alle Prozesse P_k mit A_ki = 0 fuer alle i
setze W_i := V_i fuer alle i
loop
    suche unmarkierte P_k mit Q_ki ≤ W_i fuer alle i
    Test: existiert so ein unmarkierter Prozess P_k?
        true → markiere P_k und setze W_i := W_i + A_ki fuer alle i
        false → break
end loop;
```

Am Ende befinden sich die unmarkierten Prozesse in einem Deadlock.

5.7.2 Deadlock Detection: Bemerkung

- Optimistische Annahme, dass Prozesse keine zusätzlichen Ressourcen für die Fertigstellung benötigen
- Wenn diese Annahme nicht stimmt, kann es später zu einem Deadlock kommen → wird allerdings beim nächsten Aufruf des Deadlock Detection Algorithmus erkannt

5.7.3 Deadlock Recovery

Zum Auflösen eines erkannten Deadlocks gibt es mehrere Herangehensweisen:

- **Abbrechen aller** betroffenen Prozesse → durch Abbruch eines Prozesses werden die Ressourcen wieder freigegeben (häufig verwendete Strategie)
- **Rollback aller** beteiligten Prozesse bis zu einem definierten Aufsetzpunkt; Zurücksetzung bis zu einem Punkt, wo Ressourcen noch nicht allokiert wurden; Deadlock kann sich hier allerdings wiederholen;
- **Abbrechen einzelner** beteiligter Prozesse, solange bis der Deadlock beseitigt ist (wiederholtes Aufrufen des Deadlock Detection Algorithmus)
- **Ressourcen schrittweise entziehen** und an andere Prozesse vergeben, bis kein Deadlock mehr vorliegt; Rücksetzen der Prozesse, denen Ressourcen entzogen werden, bis zum Punkt der Zuweisung

Eine Ressource kann nicht einfach wieder freigegeben werden, da davon ausgegangen werden muss, dass ein Prozess, der diese Ressource allokiert hat auch verändert und verwendet hat → Ressourcen müssen zurückgesetzt werden → Zustand des Speichers muss vor dem vergeben einer Ressource gespeichert werden
Auswahl des zu unterbrechenden Prozesses (mit welchem Prozess soll das Recovery beginnen):

- wenigste bisher verbrauchte CPU-Zeit?
- geringste Anzahl an bisher belegten Ressourcen?
- geringster Fortschritt?

5.8 Integrierte Deadlock Strategie

Kombination der genannten Ansätze

- Gruppieren der Ressourcen in Klassen und Ordnen der Klassen
  - Swappable Space (Sekundärspeicher)
  - Prozessressourcen (I/O Geräte, Files, etc.)
  - Hauptspeicher
- Circular Wait Prevention zwischen Klassen
- Verwendung der best-geeigneten Strategie in den einzelnen Klassen
  - Prozessressourcen → Deadlock Avoidance
  - Hauptspeicher → Deadlock Prevention: Preemption

5.9 Zusammenfassung

- 4 Bedingungen (notwendig und hinreichend)
  1. Mutual Exclusion
  2. Hold and Wait
  3. No Preemption
  4. Circular Wait
- Deadlock Prevention (Einschränken der Bedingungen)
  - Indirect Deadlock Prevention
  - Direct Deadlock Prevention
- Deadlock Avoidance (Keine Einschränkung der Bedingung)
- Deadlock Detection und Recovery

6 Memory Management

- Speicher ist die zentrale Ressource, die jeder Prozess benötigt
- Effektive Aufteilung und Verwaltung des Arbeitsspeichers für das Betriebssystem und Programme
- Anforderungen:
  - **Partitioning**: Speicheraufteilung auf Prozesse
  - **Relocation**: Positionierung von Code und Daten im Speicher; Virtual Memory Management
  - **Protection**: Speicherschutz; Prozesse dürfen nur auf eigene Daten zugreifen
  - **Sharing**: Gemeinsamer Zugriff auf Speicher
  - **Performance**: effektive logische / physische Organisation; Prozessorleistung soll nicht für die Speicherverwaltung verbraucht werden

6.1 Paritionierung des Speichers

Ist ein einfacher Ansatz für die Speicherverwaltung wo **Prozesse als Ganzes im Hauptspeicher** abgelegt werden. Außerdem ist das die Grundlage für moderne Speicherverwaltungmodelle:

- **Fixed partitioning** (fixe Größe)
- **Dynamic partitioning** (unterschiedliche Größe)
- **Buddy System**

6.1.1 Fixed Partitioning

Selbst innerhalb des Fixed Partitioning gibt es nochmal Unterschiede. Der Speicher kann entweder in **gleiche große Partitionen** unterteilt werden:

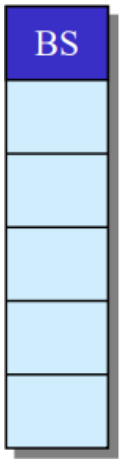

Abbildung 26: gleiche Partitionsgrößen

oder in **unterschiedlich große Partitionen**:

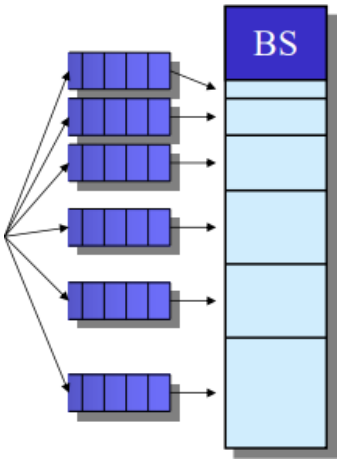

Abbildung 27: eine Process-Queue pro Partition

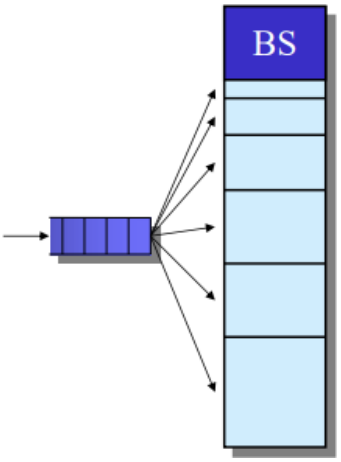

Abbildung 28: eine Process-Queue

Unterschiedlich große Partitionen können den Speicher optimaler ausnutzen als gleichbleibend große Partitionen. Kleine Prozessen werden dann in den kleinen Partitionen abgelegt, und größere in den größeren. Wie in der Abbildung 27 gut zu sehen gibt es in diesem Fall für jede Partition eine Warteschlange für die Prozesse. Das bedeutet jeder Prozess reiht sich in der Warteschlange der jeweiligen Partition ein, die am besten für ihn passt. Wenn eine Partition gerade befüllt ist, müssen die Prozesse in der Queue warten. Dieses System mit eigenen Queues pro Partition funktioniert sehr gut, wenn die Prozesse in unterschiedlichen Größen gleich oft vorkommen. Sobald es überwiegend kleine oder große gibt ist das System mit nur einer Warteschlange besser wo immer die erstbeste passende Partition befüllt wird. Dadurch wird der Speicher nicht ganz so gut ausgenutzt aber es können besser mehrere Prozesse gleichzeitig laufen.

- Unterteilung des Hauptspeichers in nicht überlappende Teile
- Prozesse, die kleiner gleich einer Partition sind, können geladen werden
- Auslagern eines Prozesses durch das Betriebssystem wenn alle Partitionen belegt sind
- Programme die größer als Partitionen sind, sind nicht so leicht ausführbar → Programmierung von *Overlays* (Programmteile des Prozesses, die austauschbar sind und dynamisch geladen werden können)

6.2 Dynamic Partitioning

Partitionsgrößen werden an die tatsächlichen Bedürfnisse der Prozesse angepasst.

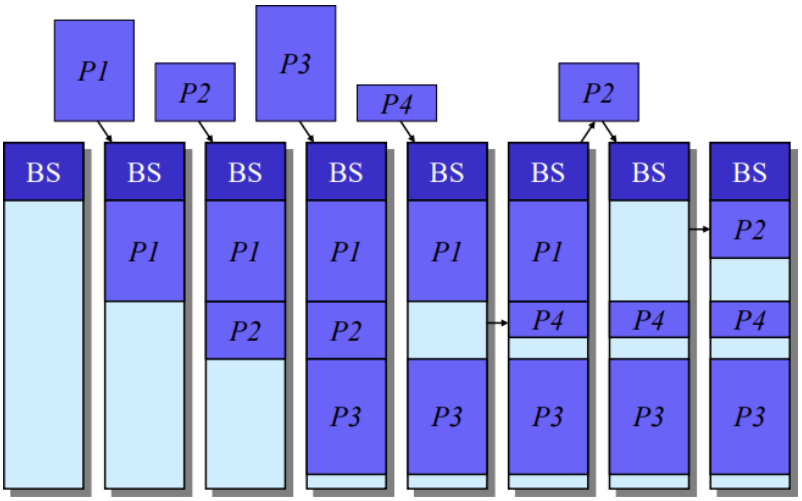

Abbildung 29: Dynamic Partitioning

In der Abbildung 29 stellen die Rechtecke $P1$, $P2$ usw. Prozesse dar. Pfeile zeigen an, wenn ein Prozess in den Speicher geladen wird. In diesem Beispiel ist nach dem Prozess $P3$ kein Platz mehr für den neuen Prozess $P4$ übrig, weshalb dann auch $P2$ ausgelagert und zu einem späteren Zeitpunkt wieder geladen wird. Um jedoch $P2$ wieder zu laden, muss $P1$ ausgelagert werden.
Bei der Dynamischen Partitionierung muss der Speicher laufend reorganisiert werden. Um den Speicher optimal verwenden zu können werden Partitionen auch immer wieder *zusammen geschoben* um freien Speicher zwischen den Prozessen zu vermeiden. Diesen Vorgang nennt man **Compaction**.

- Partitionen variabler Länge und Anzahl
- Zwischen Partitionen entstehen Löcher → **Compaction**: Verschieben der Partitionen um statt Löchern größeren zusammenhängenden freien Speicherbereich zu erhalten
- Placement Strategies:
  - **Best fit**: Speicherblock, der am wenigsten Freien Speicher übrig lässt wird genommen.
  - **First fit**: Erster Speicherblock, der passt wird genommen.
  - **Next fit**: Zuerst wird der erste Speicherblock, der passt genommen, sobald der nächste Prozess geladen werden soll, fängt man dort an zu suchen, wo man davor aufgehört hat.

6.3 Fragmentierung des Speichers

Wie bereits beim Dynamic Partitioning kennengelernt, kann es dort passieren, dass kleine freien *Speicherschnipsel* zwischen den belegten Partitionen über bleiben. Dieser freie Speicher kann dann nicht mehr wirklich verwendet werden. Dieses Phänomen wird Fragmentierung genannt.

- **Interne Fragmentierung**: Verschwendung von Speicherplatz innerhalb einer Partition; Daten und Programme füllen die Partition nicht aus; nicht nutzbarer Speicher in den Partitionen (z.B. bei der fixen Partitionierung)
- **Externe Fragmentierung**: Zerstückelung des Speicherbereichs außerhalb der Partitionen; nicht nutzbarer Speicher zwischen den Partitionen (z.B. bei der dynamischen Partitionierung)

6.4 Buddy System

- Das Buddy System versucht die Nachteile der beiden vorher angeführten Strategien zu verringern.
- Zerlegung des Speichers in *Buddies* der Größe 2^k (Bytes oder Words) mit MinVal $\leq k \leq$ MaxVal. Wobei MinVal die Minimale Buddy Größe bestimmt und MaxVal analog die maximale Größe.

6.4.1 Vorgangsweise

1. Start: Anlegen eines Blocks maximaler Größe
2. Anforderung eines Blocks der Größe S:
   (a) Allokiere einen Block der Größe S^U, wobei $S^{U-1} < S \leq 2^U$; 2^U ist die nächst größere Block-Größe der geforderten Block-Größe; Es wird also ein 2^U gesucht, welches größer gleich der angeforderten Byte bzw. Word Anzahl ist.
   (b) gibt es keinen solchen Block, zerteile den nächstgrößeren Block in zwei kleinere; wenn es z.B. keinen Block der Größe 2^k gibt, wird ein Block der Größe 2^{k+1} genommen und halbiert
   (c) wiederhole ggf. den vorherigen Schritt
3. Bei Freiwerden eines Blocks kombiniere ursprüngliche zusammengehörige Blöcke

6.4.2 Beispiel

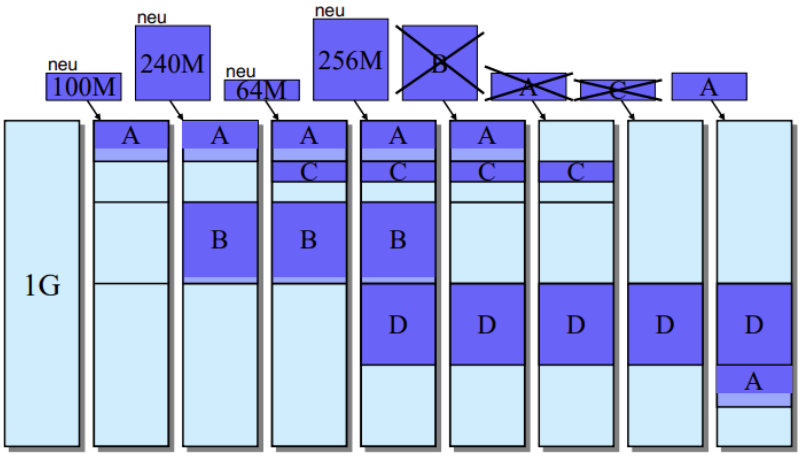

Abbildung 30: Buddy System Beispiel

In diesem Beispiel steht das G für *Gigabyte* und das M für *Megabyte*. Anfänglich gibt es einen freien Speicher der Größe von einem Gigabyte. Es gibt also nur einen Buddy, welcher den gesamten Speicher abdeckt. *(Schritt 1)*
Dann soll ein Prozess der Größe 100 Megabyte eingelagert werden. Die nächst höhere Zweierpotenz wäre $2^7 = 128$, das bedeutet es wird nach einem freien Buddy dieser Größe gesucht. *(Schritt 2a)* Allerdings gibt es keinen Buddy in dieser Größe. Deswegen wird ein Buddy der nächst höhere Zweierpotenz $2^{7+1} = 256$ gesucht. Auch diesen gibt es nicht. Dieser Schritt wird so oft wiederholt bis ein passender Buddy gefunden wird. Der Buddy, der gefunden wird, ist in diesem Beispiel der Anfangs-Buddy, welcher den gesamten Speicher abdeckt. Nachdem immer Buddies, welche größer waren, als angefordert gesucht wurden, werden diese jetzt jeweils halbiert. *(Schritt 2b und 2c)*
Die kleinste Zerteilung kann jetzt für den ersten Prozess verwendet werden. Der nächste Prozess findet bereits einen Buddy in der richtigen Größe und kann direkt geladen werden. *(Schritt 1a)*
Wenn jetzt dann der Prozess B terminiert, wird der Speicher freigegeben. Sobald es benachbarte, kombinierbare Buddies gibt, werden diese zusammengefügt. In diesem Fall können bei der Terminierung von B keine Buddies zusammengefasst werden. Durch das Terminieren von C kann allerdings zuerst der Buddy oberhalb von C mit dem Buddy, wo C früher abgespeichert wurde kombiniert werden, und dann der resultierende Buddy mit dem Buddy wo B vorher abgespeichert wurde. *(Schritt 3)*

6.4.3 Anmerkung

Wie auch beim Beispiel gut zu sehen ist, kann derselbe Prozess auf unterschiedlichen Orten im Speicher abgelegt sein. Beispielsweise ist Prozess A im ersten Durchlauf ganz am Anfang des Speichers abgelegt worden, als er jedoch ein zweites mal ausgeführt wurde ganz am Ende des Speichers. Das bedeutet festgeschriebene Adressen zu Daten und Instruktionen müssen an verschiedenen Speicherstellen positioniert werden können, ohne das ein Fehler entsteht.

6.5 Relocation

- Der von Prozessen belegte Speicherbereich ist nicht statisch fixiert (Aufruf bei unterschiedlicher Speicherbelegung, Swapping, Compaction)
- Instruktionen, Daten, usw. müssen an verschiedene Speicherstellen positioniert werden können.
- **Referenzen auf physischen Speicher müssen veränderbar sein** → Unterscheidung zwischen logischer Adresse und physischer (absoluter) Adresse

6.6 Speicheradressierung

- **Physische (absolute) Adresse**: absolute, tatsächliche Position im Hauptspeicher
- **Logische Adresse**: Referenz einer Position im Speicher, unabhängig von Organisation des Speichers; Wie ein Name für einen Speicherort; (z.B. Variablen im Code sind nur Namen für einen Speicherort, wo der Wert tatsächlich drinnen steht, es ist nicht bekannt und nicht wichtig zu wissen wo genau diese Werte im Speicher stehen.)
- **Relative Adresse**: Position relativ zu einem bekannten Punkt im Programm (logische Adresse)
- Übersetzter Code enthält logische, meist relative Adressen → zur Laufzeit: **Adressübersetzung** auf absolute Adressen

6.6.1 Einfache Adressübersetzung

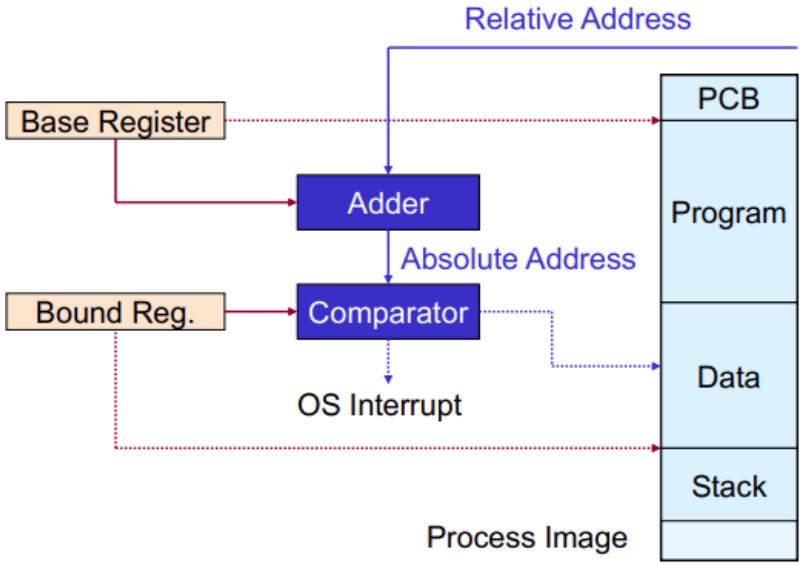

Abbildung 31: Einfache Adressübersetzung

- Adressübersetzung durch Hardware
- Physische Startadresse des laufenden Prozesses im **Base Register**
- Endadresse des physischen Speichers des Prozesses im **Bound Register**
- Zugriff auf relative Adressen: Inhalt des Basisregisters wird zur relativer Adresse addiert
- Speicherschutz (Protection): Überprüfung, ob die Adresse im gültigen Bereich (innerhalb der Bound) liegt

6.6.2 Segmentierung

- **Segmente**: Unterteilung von Programmen in Blöcke unterschiedlicher Länge
- Programme müssen durch Segmentierung nicht mehr als ein großer Block im Speicher geladen sein → Blöcke im Speicher müssen nicht zusammenhängen → bessere Nutzung des Speichers
- Segmente enthalten Code und / oder Daten
- Beim Laden des Prozesses werden seine Segmente *beliebig* im Speicher platziert
- keine interne Fragmentierung
- aber externe Fragmentierung immer noch möglich
- sichtbar für den Programmierer (Programmierer muss diese definieren)
- **Segment Table** pro Prozess, um die Segmente finden zu können; für jedes Segment existiert ein Eintrag (Start, Länge)

Segmentierung: Adressübersetzung Jede logische Adresse besteht jetzt aus einer Segment Nummer und einem Offset. Die Startadresse (Base) eines Segments wird zum Offset dazu addiert um die physische Adresse zu erhalten. Speicherschutz wird durch das Überprüfen der physischen Adresse mit der Länge des Segments garantiert.

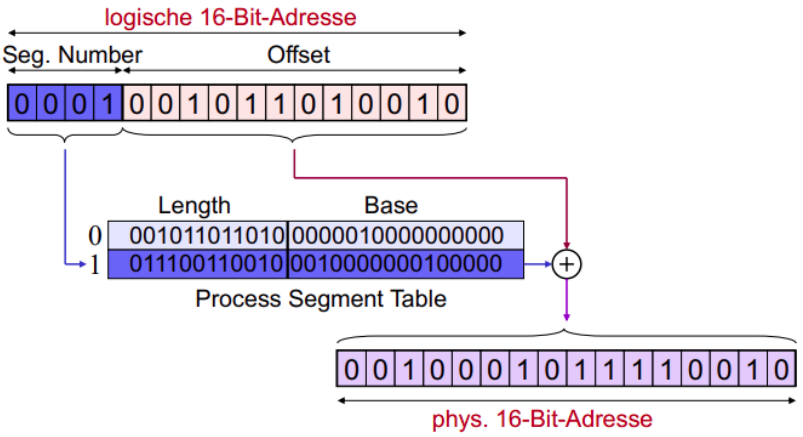

Abbildung 32: Segmentierung Adressübersetzung

6.6.3 Paging

Händisches Segmentieren ist aufwändig → automatisiertes Paging als Lösung

- **Frames**: Unterteilung des Hauptspeichers in Seitenrahmen gleicher Größe; Ähnlich wie bei der Fixen Partitionierung
- **Pages**: Prozesse werden in Seiten der selben Größe wie die Frames unterteilt
- Seiten werden in Seitenrahmen geladen
- **Page Table** pro Prozess enthält für jede Page die aktuelle Frame-Nummer
- **Free Frame List** verweist auf freie Frames im Speicher
- Pages sind sehr klein → fast immer kleiner als Segmente

Paging: Seitengröße Eine Seitengröße ist immer 2^k Bytes oder Words, damit die Seiten immer bei einer physischen Adresse mit 2^k beginnen. Das bedeutet die niederwertigen k Bits beschreiben den Offset innerhalb einer Seite, während die höherwertigen $n-k$ Bits die Seitennummer beschreiben → jede Adresse zerfällt in zwei Teile: Seitennummer und Offset. Der Offset einer logischen Adresse ist immer derselbe in der physischen Adresse. Die Pagenummer muss allerdings immer durch die Framenummer ersetzt werden, um zur physischen Adresse zu gelangen.

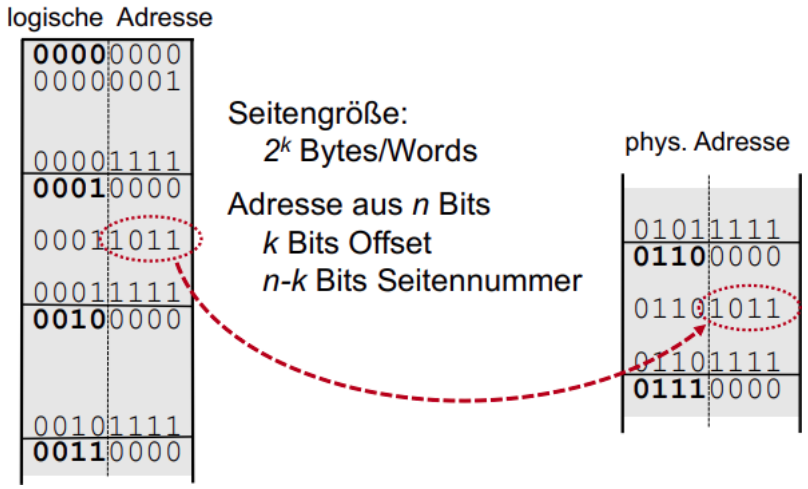

Abbildung 33: Paging Seitengröße

Paging: Adressübersetzung Aus dem Page Table wird die Framenummer herausgelesen und mit dem Page-Nummer-Teil der logischen Adresse ersetzt umm auf die physische Adresse zu kommen.

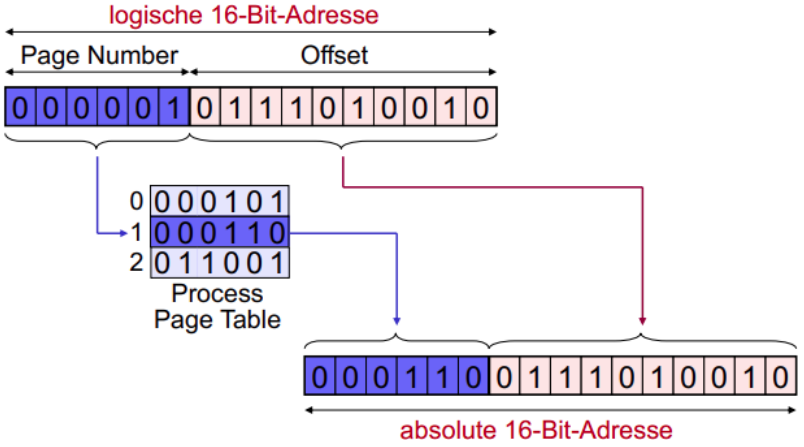

Abbildung 34: Paging Adressübersetzung

Paging: Logische Adressen

- Logische Adresse aus zwei Teilen (Page Number, Offset)
- Mit Hilfe der Page Table ermittelt der Prozessor die absolute Adresse: (Frame Number, Offset)
- Seitengröße: 2^k Bytes oder Words
  - Seitenunterteilung für Tools (Compiler, Linker) bzw. Programmierer *unsichtbar*
  - Adressen einfach zu übersetzen

6.7 Virtual Memory

- Dynamische Adressübersetzung
- Aufspaltung von Prozessen in Seiten (oder Segmente), die nicht hintereinander im Speicher abgelegt werden müssen → Prozesse müssen sich nicht als ganzes im Speicher befinden
- nur **aktuell benötigte** Seiten im Speicher
  - mehr Prozesse passen in den Hauptspeicher
  - einzelne Prozesse können größer als der Hauptspeicher sein

- Logische Adressen referenzieren Virtual Memory
- Seitentabelle (Segmenttabelle) und Memory Management Hardware unterstützen die Übersetzung von virtuellen Adressen in absolute Adressen
- **Resident Set**: Teile eines Prozesses, die sich im Hauptspeicher (RAM) befinden
- **Page Fault**: Ausnahmebehandlung, wenn eine Adresse im VM referenziert wird, die sich nicht im Hauptspeicher befindet
  - **Demand Paging**: entsprechender Speicherbereich wird von Sekundärspeicher geladen
- **Thrashing**: Durch häufige Page Faults verbringt der Prozessor sehr viel Zeit mit dem Laden vom Sekundärspeicher → drastischer Einbruch der Effektivität (z.B., Resident Set eines Prozesses zu klein)

6.7.1 Lokalitätsprinzip

Es werden nie alle Pages eines Programms gebraucht, da Programme recht lokal abgearbeitet werden. Ab und zu gibt es große Sprünge, wenn zum Beispiel ein Unterprogramm aufgerufen wird, allerding bleiben dann die Bearbeitung lokal in diesem Unterprogramm → nur die lokalen notwendigen Pages müssen eigentlich im Arbeitsspeicher sein. Die Strategie, dass nur die benötigten Teile eines Programms im Speicher auffindbar sein müssen wird **Virtual Memory Management** genannt.
Das Lokalitätsprinzip beschränkt sich nicht nur auf den Speicher, sonder hat eine vielzahl an Anwendungen und erstreckt sich auch über die zeitliche Dimension.

6.7.2 Paging

- Pro Prozess eine Seitentabelle
  - Tabellengröße hängt von der Seitengröße und der Prozessgröße ab
- Tabelleneinträge in Page Table
  - **Present Bit**: Ist diese Seite im Hauptspeicher?
  - **Frame Number**: wo Seite zu finden ist
  - **Modified Bit**: Wurde die Seite seit dem Laden verändert?
  - **Control Bits**: r/w-Bits, Locking, Kernel vs. User Page

6.7.3 Paging: Adressübersetzung

Der Page Table Pointer Zeigt auf den Page Table im Speicher. Sobald das Present Bit gesetzt ist, kann die Frame Number aus dem Page Table ausgelesen werden. Die Page Table Größe ist proportional zur Prozess-Größe. Wenn mehrere Prozesse gleichzeitig geladen sind, wird viel Speicherplatz für die Page Tables verwendet → zwei Lösungen:

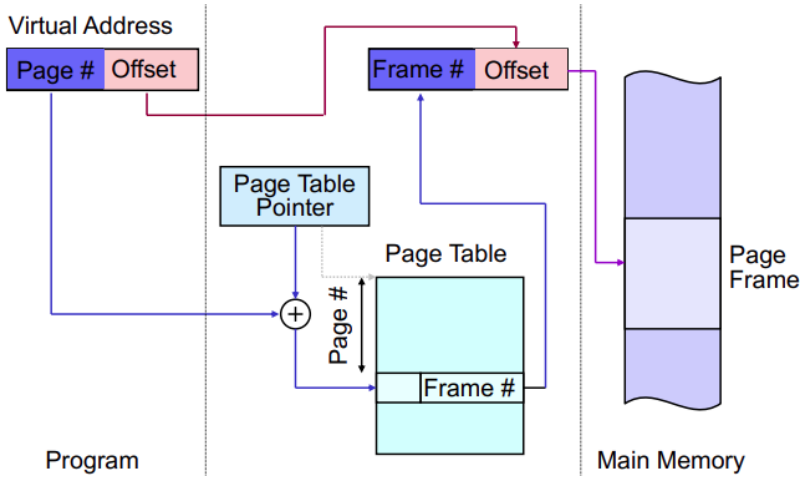

Abbildung 35: Paging Adressübersetzung

- Multilevel Page Tables 6.7.4
- Inverted Page Table (IPT) 6.7.5

6.7.4 Multilevel Page Tables

Bei großen Prozessen beenötigt der Page Table viel Speicherplatz, der sogar über mehrere Seiten gehen kann. → nur Teile des Page Tables sollten im Arbeitsspeicher gehalten werden. Die Page

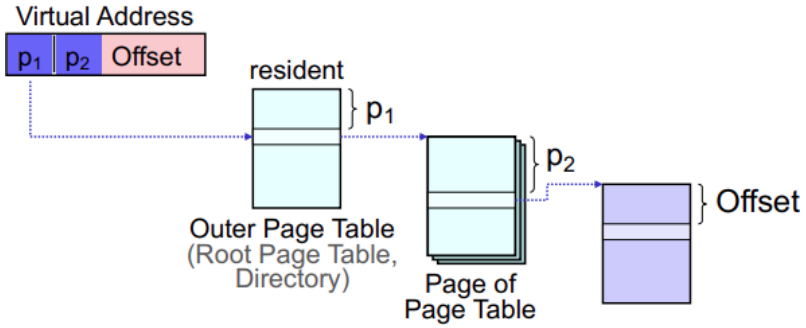

Abbildung 36: Multilevel Page Table

Number wird noch einmal in zwei Teile unterteilt p_1 und p_2. Die höherwertige Page Number p_1 wird verwendet um im Outer Page Table die Seiten zu finden. Der Outer Page Table ist wie eine Indexierung aller Pages (Inhaltsverzeichnis mit Seitenangabe). Der Outer Page Table speichert also Verweise auf die Seiten des Page Tables, welche entweder im Arbeitsspeicher oder ausgelagert sein können. Innerhalb der Page Tables kann dann mit p_2 auf die richtige Page zugegriffen werden. Der Outer Page Table muss immer im Arbeitsspeicher bleiben.

6.7.5 Inverted Page Table (IPT)

Statt für jeden Prozess einen Page Table zu machen, wird ein Page Tabel für den Arbeitsspeicher angelegt in dem es für jeden Frame im Speicher einen Eintrag gibt. Für jeden Frame kann so herausgefunden werden welche Seite von welchem Prozess dort gerade geladen ist.

- Ein IPT für das gesamte System
- Ein IPT-Eintrag pro Frame
  - **Eintrag muss im IPT gesucht werden**: assoziative Suche in IPT bzw. Hashing um Suche in IPT zu beschleunigen
  - **Page Fault**: gesuchter Eintrag nicht im IPT
- IPT ist proportional zur Größe des Arbeitsspeichers

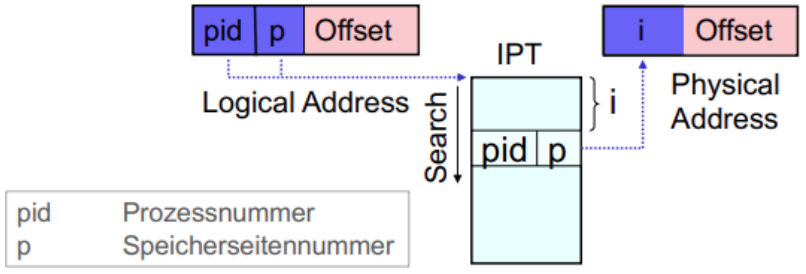

Abbildung 37: Inverted Page Table

Ein Problem ist das Aufwändige Suchen in der Tabelle, was bei jedem Speicherzugriff anfällt → Lösung: Translation Lookaside Buffer 6.7.6

6.7.6 Translation Lookaside Buffer

Pro Zugriff auf das Virtual Memory sind mindestens zwei Zugriffe auf den phsysischen Speicher notwendig → Zugriffe werden gecached.

- Übersetzungsinformationen werden in einem Cache in der CPU abelagert
- **Translation Lookaside Buffer (TLB)**: Cache für Einträge der Seitentabelle
  - Einträge enthalten Nummern (Page, Frame) der zuletzt verwendeten Seiten
  - Assoziativer Zugriff beim Suchen
  - Löschen des TLB bei jedem Context Switch 2.5
  - kann nur sehr wenige Zugriffe speichern (ca. 4 - 8 Stück)

Translation Lookaside Buffer: Adressübersetzung Bei der Adressauflösung wird zuerst der Cache *befragt*. Bei einem sogenannten **TLB hit** wurde die gesuchte Adresse direkt im Cache gefunden, und kann verwendet werden. Bei einem **TLB miss** muss im Page Table nach dem passenden Eintrag gesucht werden. Wenn der Eintrag gefunden wurde wird zuerst überprüft, ob das Present Bit gesetzt ist. Falls ja, kann diese Adresse im Cache gespeichert werden (wird durch die Hardware realisiert) und verwendet werden. Wenn nicht kommt es zu einem **Page Fault** und die Page muss erst in den Arbeitsspeicher geladen werden. Dadurch kommt es zu einer I/O Operation und demnach sehr wahrscheinlich zu einem Process Switch 2.5. Wenn dieser Prozess dann allerdings wieder zur CPU kommt, sollte die gesuchte Page im Arbeitsspeicher vorliegen und es kommt zu keinem Page Fault mehr.

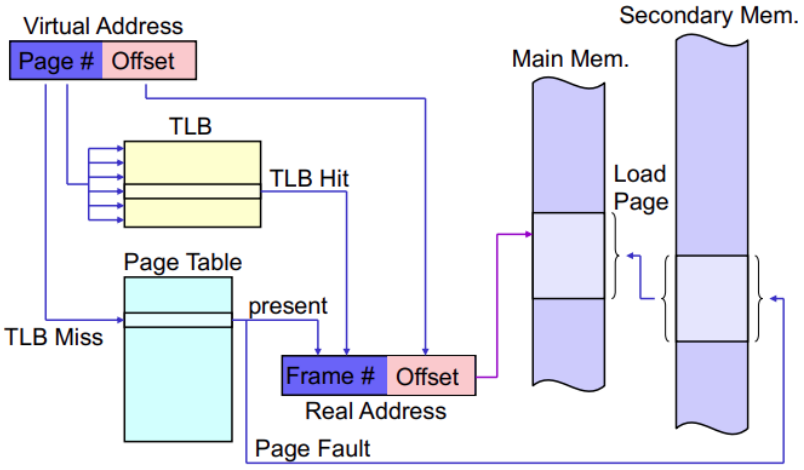

Abbildung 38: Translation Lookaside Buffer Adressauflösung

6.8 Fetch Policy

Die Fetch Policy bestimmt, wann eine Seite geladen wird.

- **Demand Paging**
  - lädt eine Seite, wenn eine Adresse der Seite referenziert wird
  - Nachteil: viele Page Faults beim Start eines Prozesses
- **Prepaging**:
  - lädt mehr Seiten als angefragt im Voraus
  - durch Lokalitätsprinzip motiviert (Prog., Disk)
  - Nachteil: Laden nicht benötigter Seiten

6.9 Replacement Policy

Die Replacement Policy bestimmt, welche Seite beim Laden einer neuen Seite im Hauptspeicher ersetzt wird.

- **lokal**: Austausch innerhalb der Seiten des Prozesses
- **global**: Austauschstrategie wird auf alle Seiten des Hauptspeichers angewandt

6.9.1 OPT Policy

Optimale Strategie (theoretische Strategie)

- erzeugt die wenigsten Page Faults
- ersetzt die Seite, deren nächste Referenz am weitesten in der Zukunft liegt (kann man nicht wissen)
- keine reale Strategie
- wird zur Bewertung anderer Strategien herangezogen

6.9.2 LRU Policy

Least Recently Used

- ersetzt die Seite, die am längsten nicht benützt worden ist
- aufgrund des Lokalitätsprinzips 6.7.1 ist zu erwarten, dass die Seite auch in naher Zukunft nicht referenziert werden wird
- Anzahl der Page Faults kaum höher als bei OPT
- Nachteil: Implementierung sehr aufwendig aufwendig; Speichern der Zugriffszeiten für Seiten, Suche nach der am längsten nicht mehr verwendeten Seite

6.9.3 FIFO Policy

First In - First Out

- bei vollem Seitenkontingent wird die älteste Seite ersetzt
- einfache Implementierung: Pointer, der zyklisch über die Seitenrahmen fortgeschaltet wird
- Nachteil: eine Seite, die gerade oft verwendet wird, kann die älteste sein und wird deswegen ausgelagert

6.9.4 Clock Policy

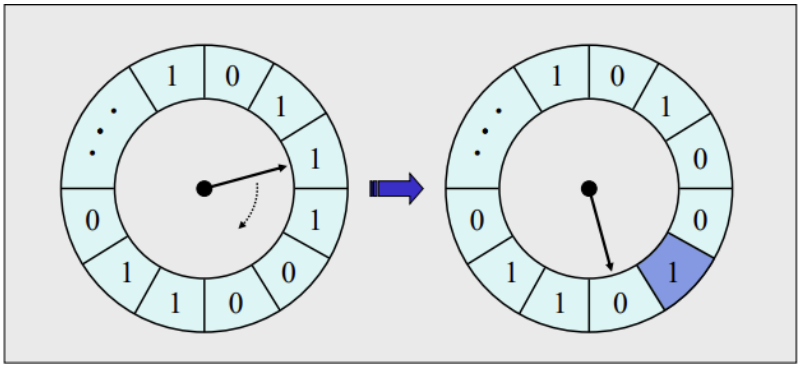

Abbildung 39: Clock Policy

- kaum mehr Page Faults als LRU aber mehr Frames pro Prozess → bessere Annäherung
- Ringpuffer mit Frames, die für Austausch in Frage kommen sowie ein Positionszeiger

- Ringpuffer ist mit Use Bits gefüllt
- nach Page Fault: Zeiger wird auf den folgenden Frame im Ringpuffer gesetzt
- Use Bit pro Frame, wird auf 1 gesetzt wenn Seite geladen oder referenziert wird
- bei Page Fault:
  - Suche erste Seite ab Zeigerposition mit Use Bit = 0
  - Setze dabei Use Bits mit Wert 1 auf 0

6.10 Größe des Resident Set

Das Resident Set ist die Menge an Seiten eines Prozesses, die sich im Hauptspeicher befinden. Es stellt sich die Frage wie viele Frames sollen dem Prozess zugeteilt werden?

- wenige Frames → viele Page Faults
- viele Frames → Einschränkung der Parallelität (wenige Prozesse im Arbeitsspeicher)
- zwei Strategien:
  - **Fixed Allocation**
  - **Variable Allocation**

6.10.1 Working Set Strategie

- Variable Allokation von Frames für einen Prozess basierend auf Lokalitätsannahme
- **Working Set** eines Prozesses zum Zeitpunkt t: $W(D,t)$ = Menge der Seiten des Prozesses, die in den letzten D virtuellen Zeiteinheiten referenziert wurden.
  - virtuelle Zeit bezieht sich auf betrachteten Prozess
  - D ... Zeitfenster (Window of Time); wie weit in die Vergangenheit gesehen werden soll
  - Funktion $W(D,t)$ beschreibt Lokalität des Prozesses
- Working Set eines Prozesses wächst beim Start schnell an
- stabilisiert sich während der weiteren Prozessausführung (Lokalitätsprinzip 6.7.1)
- wächst wieder, wenn sich die Abarbeitung in einen anderen Adressbereich verschiebt (Aufruf von Unterprogramm)

Ermittlung der Größe des Resident Set

- Beobachtung des Working Set der Prozesse
- Periodisches Löschen der Seiten, die sich nicht im Working Set befinden
- Wenn Resident Set $\subset$ Working Set
  - allokiere fehlende Frames
  - sind nicht genügend Frames verfügbar, suspendiere den Prozess und probiere Allokation später nochmals

Probleme bei der Realisierung

- viel Protokollarbeit
- Mitloggen der Seitenreferenzen
- Ordnen der Seitenreferenzen
- Wie groß soll D sein?; müsste eigentlich dynamisch angepasst werden, je nachdem was der Prozess gerade macht
- optimales D ist zur Laufzeit unbekannt und variiert

Die gängigere Praxis ist daher das Beobachten der Anzahl an Page Faults pro Zeitintervall und Prozess statt Working Set.

6.11 VM und Paging Protection

- Über die Adressübersetztung können nur gültige Adressen berechnet werden
- Verfügbarkeit von linearen, kontinuierlichen virtuellen Adressbereichen für jeden Prozess
- Adressbereiche werden entweder mittels Seitentabelle auf physischen Speicher abgebildet oder sind *protected*
- bei jedem Zugriff auf eine (beliebige) Adresse
  - entweder es gibt einen Verweis auf eine physische Adresse in der Page Table
  - oder es kommt zum Page Fault: wenn es sich nicht um eine ausgelagerte Seite handelt, liegt eine Speicherbereichsverletzung vor

6.11.1 Protection Keys

Frames und Prozesse bekommen jeweils einen Schlüssel. Sobald es zu einem Speicherzugriff kommt wird durch die Schlüssel überprüft, ob das erlaubt ist.

Variante 1

- Pro physischem Frame gibt es einen Protection Key
- Jeder Prozess hat einen Protection Key, der beim Speicherzugriff mit dem Key des Frames verglichen wird: Ungleichheit triggert einen Fehler

Variante 2

- Pro TLB 6.7.6 Eintrag gibt es einen Key (bzw. Access ID)
- Jeder Prozess hat eine Menge von Protection Key Registers, die bei Speicherzugriff mit TLB Key verglichen werden
- Fehler, wenn kein Key gleich dem TLB Key (Bsp.: Itanium Prozessor)

6.12 Zusammenfassung

- **Partitionierung** des Hauptspeichers
  - Aufteilung auf mehrere Prozesse
  - Partitionen oder Virtual Memory Management
  - es wird nur eine Teilmenge des Prozesses im Speicher benötigt
- **Relocation**: logische Adressierung benötigt
- **Virtual Memory**
  - logischer Speicher ¿ physischer Speicher
  - Paging (Segmentierung)
  - Adressübersetzungstabellen und -hardware
  - Lade- und Austauschstrategien
  - Wahl der Größe des Resident Set
  - Protection

7 Scheduling

Beim Scheduling geht es im Grunde darum die CPU Zeit aufzuteilen und an die Prozesse zu verteilen.

- Bestimmt die Abarbeitungsreihenfolge der Prozesse auf dem Prozessor
- Verschiedene Optimierungsziele:
  - Durchsatz
  - Prozessorauslastung
  - Fairness
  - Response Time
  - Einhalten von Deadlines
  - usw.
- Vorgaukeln der Parallelität durch schnelles wechseln der Prozesse

7.1 Schedulingebenen

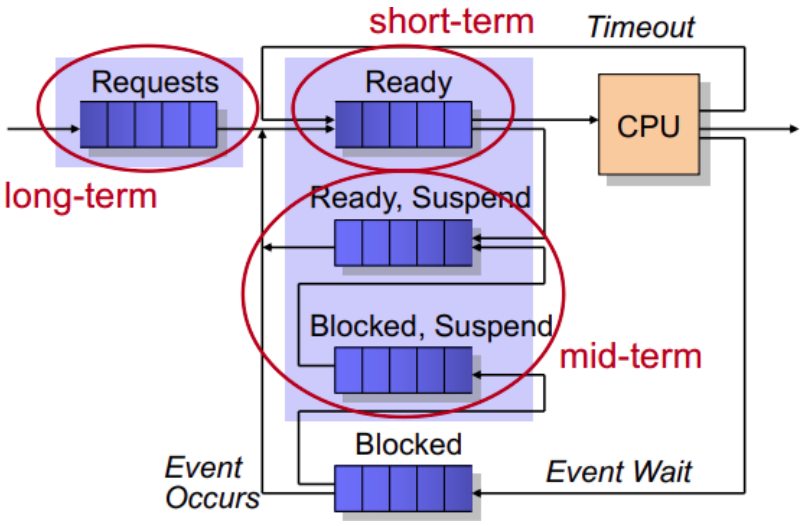

Abbildung 40: Schedulingebenen

Die Schedulingebenen hängen sehr stark mit den Prozesszuständen (Abbildung 11) zusammen.

- **Long-Term** Scheduling
  - Kreierung von Prozessen
  - bestimmt Grad der Parallelität
  - Mix von CPU- und I/O-intensiven Prozessen
  - Prozesse im New Zustand müssen nicht direkt abgearbeitet werden
- **Medium-Term** Scheduling
  - Ein- und Auslagern von Prozessen (Memory Management)
- **Short-Term** Scheduling (Dispatching)
  - bestimmt nächsten Prozess zur Ausführung
  - Bearbeiten der Ready-Queue
  - Unterbrechen von Prozessen

7.2 Short-Term Scheduling

- Welcher Prozess soll als nächstes ausgeführt werden?
- CPU-Scheduler oder Dispatcher
- Aktivierung des Dispatchers, wenn Prozessumaschaltung 2.5 angebracht sein kann
  - System Calls
  - Traps
  - I/O Interrupt, Signale
  - Clock-Interrupt (Unterbrechen von Prozessen, die schon lange genug die CPU in Anspruch hatten)

7.3 Scheduling Kriterien

	User-Oriented	**System-Oriented**
Performance	Response Time, Turnaround Time, Deadlines	Throughput, Processor Utilization
Other	Predictability	Fairness, Ressource Balance, Priorities

In diesem Fall bezieht sich User-Oriented auf alle User-Prozesse. Response Time bedeutet, dass erwartet wird, dass ein Prozess innerhalb einer vorgeschriebenen Zeitspanne ein Ergebniss liefert. Ein Beispiel für Predictability wäre ein Wecker-Programm, oder etwas was exakt zu einem Zeitpunkt ausgeführt werden soll.

7.4 Verwendung von Prioritäten

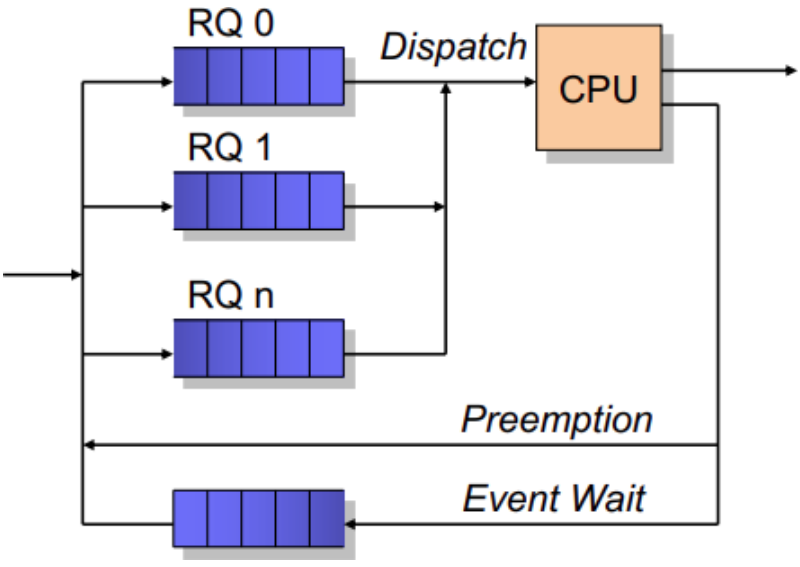

Abbildung 41: Prioritäten

Ein Beispiel wäre statt einer Ready-Queue mehrere zu realisieren, jede mit einer eigenen Priorität. Wenn die Warteschlange mit der höchsten Priorität RQ_0 leer ist, wird in der nächsten Warteschlange RQ_1 nach Prozessen geschaut usw. Der Nachteil ist, dass **Starvation** möglich ist. Diese Prioriätsstrategie ist sehr leicht umsetzbar, allerdings nicht optimal.

7.5 Scheduling Strategien

Bei Scheduling Strategien können zwei Parameter unterschieden werden:

- **Selection Function**: Auswahl des nächsten auszuführenden Prozesses
  - bisherige Verweildauer
  - bisherige oder gesamte Ausführungszeit
  - Fertigstellungszeitpunkt (Deadline)
- **Decision Mode**
  - **Non-Preemtive**: keine externe Unterbrechung von Prozessen durch das Betriebssystem
  - **Preemtive**: Unterbrechung von Prozessen durch Betriebssystem ist möglich

7.5.1 Scheduling Vokabular

- **Normalized Turnaround Time**: mittlere Zeit die ein Process im System verbringt
- **Service Time**: Abarbeitungszeit eines Prozesses

7.5.2 Task Set für Beispiele

In Folge werden verschiedene Scheduling Strategien genauer beleuchtet. Für die verschiedenen Ansätze wird immer dassselbe Task Set verwendet:

Prozess	Arrivial Time	Service Time
$P1$	0	3
$P2$	2	6
$P3$	4	4
$P4$	6	5
$P5$	8	2

Wobei die Arrivial Time und Service Time keiner bestimmten Einheit folgen.

7.5.3 First Come First Served (FCDS)

- **Selection function**: Auswahl des Prozesses, der bereits am längsten in der Ready Queue verweilt
- **Decision Mode**: non-preemptive

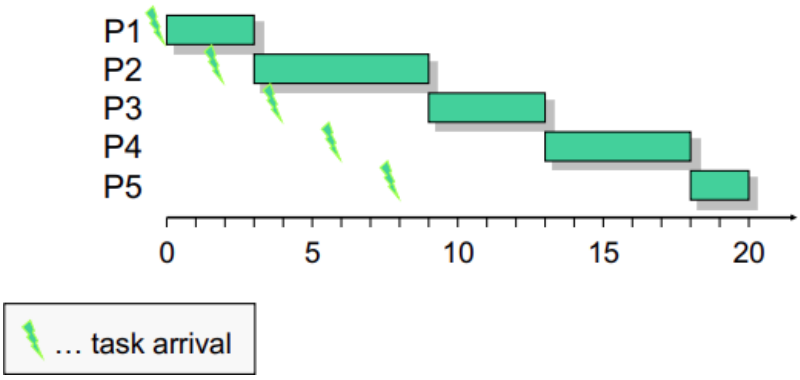

Abbildung 42: First Come First Served Beispiel

Eigenschaften

- begünstigt:
  - lange Prozesse (Normalized Turnaround Time); Lange Prozesse bekommen im Vergleich zur Wartezeit mehr CPU Zeit
  - CPU-intensive Prozesse (CPU-Monopolisierung durch Prozesse ohne I/O)
- schlechte Auslastung von CPU und I/O
- selten *pures* FCFS

7.5.4 Round Robin (RR, Time Slicing)

- **Selection function**: Auswahl des Prozesses, der bereits am längsten in der Ready Queue verweilt (wie FCFS)
- **Decision Mode**: preemptive
- Zeitscheiben gleicher Länge werden zyklisch an Prozesse vergeben (Clock-Interrupt)

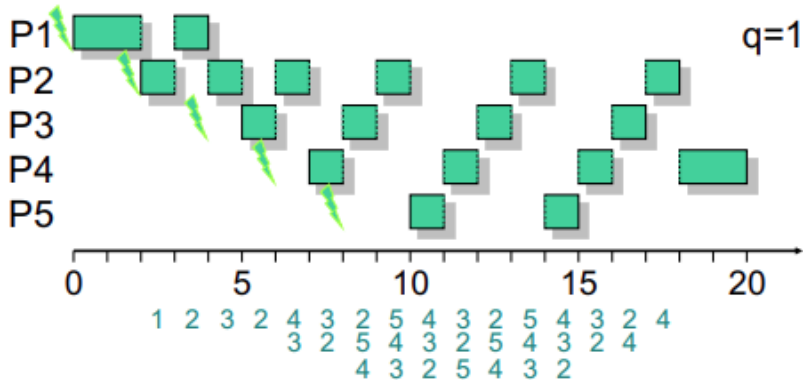

Abbildung 43: Round Robin Beispiel

Eigenschaften

- Sinnvolle Zeitscheibenlänge:
  - viel länger als Clock Interrupt + Scheduling (Process Switch 2.5)
  - etwas länger als eine typische Interaktion
  - Verwaltungs-Overhead
  - zu viele Process Switches verlangsamen die Abarbeitung
- lange Prozesse werden zerstückelt → kurze Prozesse müssen nicht so lange warten wie beim FCFS
- Benachteiligt I/O-intensive Prozesse
  - Wenn ein Prozess *nur rechnet* kann er seine Zeitscheibe voll ausnutzen.
  - Wenn ein Prozess viel I/O tätigt, wird mehr gewartet, als gearbeitet.
  - I/O-intensive Prozesse schöpfen Zeitscheiben nicht voll aus und werden während des Blockierens von CPU-intensiven Prozessen *überholt*

Verbessertes Round Robin ist das sogenannte **Virtual Round Robin**.

Virtual Round Robin Es wird eine neue Queue eingefügt, die **Auxiliary Queue** worin I/O-intensive Prozesse eingereiht werden. Die Auxiliary Queue wird dann der normalen Ready Queue bevorzugt, um den I/O-intensiven Prozessen Zeit zum Aufholen zu geben.

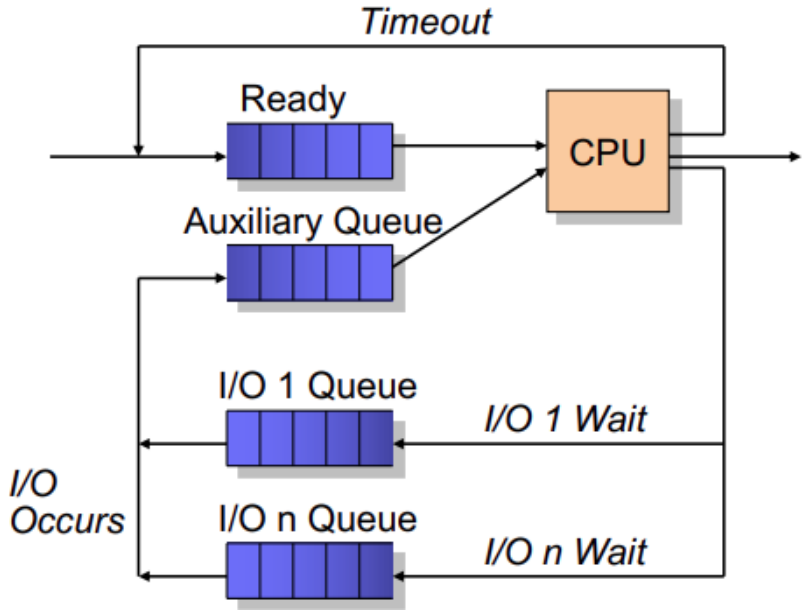

Abbildung 44: Virtual Round Robin

7.5.5 Shortest Process Next (SPN)

- **Selection function**: Prozess mit kürzestem erwarteten CPU-Burst zuerst
- **Decision Mode**: non-preemptive

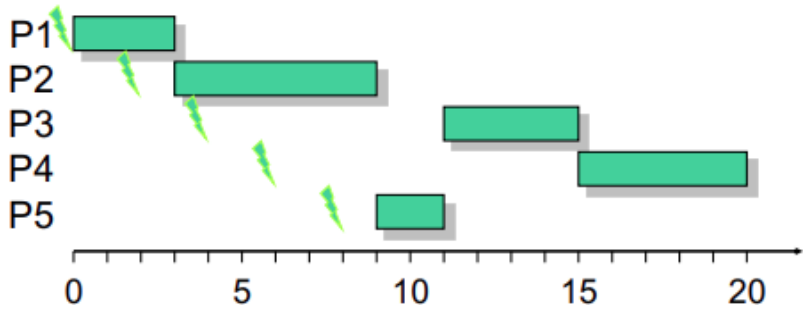

Abbildung 45: Shortes Process Next Beispiel

Eigenschaften

- bessere Response Time als FCFS
- größere Variabilität der Response Times (Verzögerung langer CPU-intensiver Prozesse)
- Nachteile:
  - Schätzung der Abarbeitungszeiten nicht wirklich möglich
  - Echtzeitprogrammierung nicht wirklich möglich (z.B. Wecker Programm)
  - Starvation langer Prozesse
  - nicht für interaktiven Betrieb geeignet (CPU-intensiver Prozess, der früh ankommt, kann CPU-monopolisieren)

7.5.6 Shortest Remaining Time SRT

- **Selection function**: Prozess mit kürzestem erwarteten CPU-Burst zuerst (wie SPN)
- **Decision Mode**: preemptive

Eigenschaften

- kürzere Prozesse werden fair behandelt
- nicht so viele Interrupts wie bei Round Robin
- Protokollieren der Service Times ist notwendig (Abarbeitungszeit muss im Vorhinein bekannt sein)
- Starvation möglich

7.5.7 Highest Response Ratio Next

- **Selection function**:
  - Response Ratio $RR = \frac{w+s}{s}$ wobei w...gesamte bisherige Wartezeit und s...geschätze Service Time
  - Auswahl des Prozesses mit größtem Response Ratio RR
  - kurze Prozesse haben in der Regel eine hohe Response Ratio
- **Decision Mode**: non-preemptive

Eigenschaften

- kürzere Prozesse werden fair behandelt
- keine Starvation
- Schätzung der Service Times notwendig
- Protokollierung der Wartezeit

7.5.8 Feedback Scheduling

- **Selection function**: basiert auf bisheriger Ausführungszeit
  - je mehr CPU Zeit ein Prozess bisher konsumiert hat, desto niedriger wird seine Priorität (dafür längere Zeitscheiben)
  - eigene Queue für jede Prioritätsstufe
- **Decision Mode**: preemptive

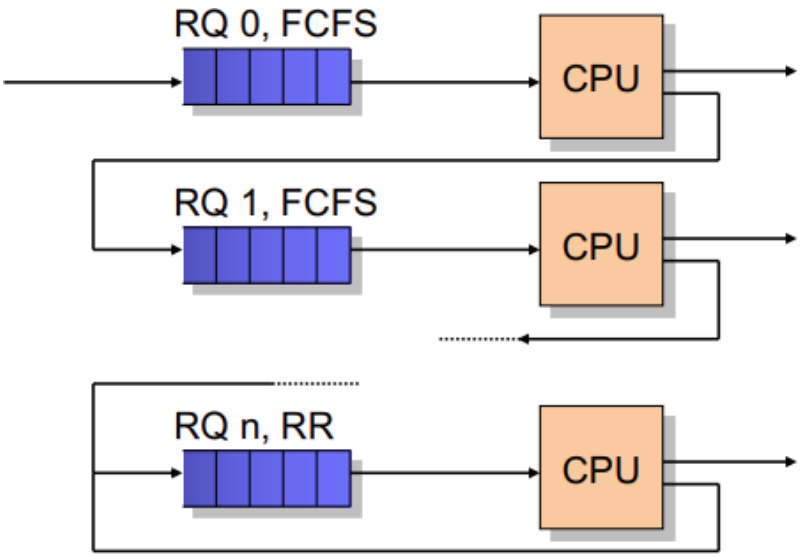

Abbildung 46: Feedback Scheduling

Eigenschaften

- als Lösung zum Schätzen der Service Time
- Starvation möglich
- Kurze Prozesse können rasch abgearbeitet werden.

Ein neuer Prozess bekommt eine hohe Priorität, aber nur kurze CPU Zeit. Wenn dieser Prozess wieder drankommt hat er bereit eine niedrigere Priorität aber mehr CPU Zeit usw.

7.6 Real-Time Scheduling

Wird in Echtzeitsstemen gebraucht um Antworten an bestimmten Zeitpunkten an die Umgebung abzugeben, zum Beispiel in Sicherheitssystemen, Elektronische Unterstützung in Autos (ABS, ESB). Es muss sichergestellt werden, dass bestimmte Prozesse zu bestimmten Zeitpunkten fertig sind und ihr Ergebnis liefern.

- Deadlines (hard vs. soft real-time)
- „fast is not real-time"(nur weil man schnell ist kann man **nicht garantieren**, dass man rechtzeitig fertig ist)
- Prozesse oft periodisch
- Scheduling typischerweise preemtive
- statisch vs. dynamische Prioritäten
- **Schedulability Test** als Überprüfung, ob alle Tasks rechtzeitig beendet werden können.

7.6.1 Earliest Deadline First (EDF)

- **Selection function**: Task mit frühester Completion Deadline zuerst
- **Decision Mode**: preemptive
- **Schedulability Test**: T_i...Periode C_i...Ausführungszeit

$$\sum \frac{C_i}{T_1} \leq 1$$

Eigenschaften

- Minimiert Anzahl der Deadline Misses (optimaler Scheduler)
- kann bis zu 100% CPU Auslastung schedulen
- Periode T_i von Prozessen muss bekannt sein
- Ausführungszeit C_i von Prozessen muss bekannt sein (Aufgabe des Programmierers)
- Periodische Prozesse (werden immer wieder ausgeführt)

7.7 Beispiel: Fixe Prioritäten vs. EDF

Das Task Set besteht aus zwei periodischen Prozessen A und B mit unterschiedlicher Abarbeitungszeit und Periode. Das Ende einer Periode ist auch gleichzeitig die Deadline dieses Prozesses, das bedeutet das Ergebnis muss am Ende einer Periode vorliegen.

Prozess	Abarbeitungszeit	Periode
A	2	4
B	5	10

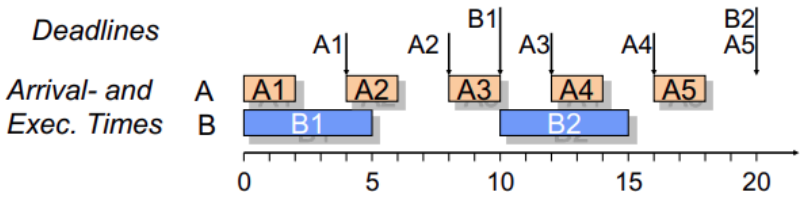

Abbildung 47: Task Set

Diese Abbildung 47 ist kein Abarbeitungsmuster, sondern zeigt lediglich die Abarbeitungszeit sowie Periode der jeweiligen Prozesse. Die Pfeile auf der Oberseite zeigen die Periodenenden und damit auch die Deadlines an.
Die Aufgabe ist es, eine CPU-Zuteilungs-Folge zu finden, sodass jeder Prozess immer zur Deadline fertig ist. Ein Beispiel wäre *Fixed Priority*, wo ein Prozess immer vorgereiht wird:

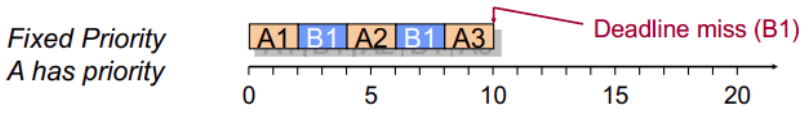

Abbildung 48: Fixed Priority, mit A

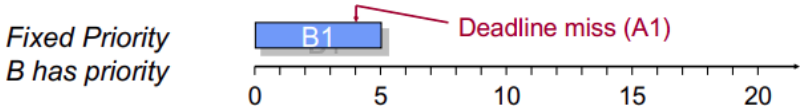

Abbildung 49: Fixed Priority, mit B

In beiden Fällen widerfahren wir einem Deadline miss. Dieses Problem lässt sich mit der EDF Strategie lösen:

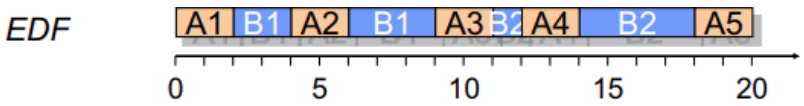

Abbildung 50: EDF Beispiel

Die Prozess Switch (2.5) Zeiten wurden für diese Beispiele Vernachlässigt.

7.8 Zusammenfassung

- Scheduling erfolgt auf verschiedenen Ebenen
  - Long-Term
  - Medium-Term
  - Short-Term
- Eine Vielzahl von Kriterien bestimmen die Wahl der Schedulingstrategie
  - Benutzerorientiert vs. Systemorientiert
  - quantitativ vs. qualitativ
- Schedulingstrategien:
  - First Come First Serve (FCFS)
  - Round Robin (RR)
  - Shortest Process Next (SPN)
  - Shortest Remaining Time (SRT)
  - Highest Response Ratio Next (HRRN)
  - Feedback Scheduling
  - Real-Time Scheduling
    - Earliest Deadline First (EDF)

8 Ein-/ Ausgabe und Disk Scheduling

8.1 Ein-/ Ausgabe

I/O ist ein sehr schwieriger Teil des Betriebssystemdesign. Es gibt unzählige verschiedene externe Geräte die nicht wirklich unter einem Hut gebracht werden können. Dennoch kann unter 3 Arten an externen I/O Geräten unterschieden werden:

- **Mensch-Interface**: z.B. Anzeige, Tastatur, Maus
- **Maschinen-Interface**: elektronische Geräte z.B. Datenlaufwerke, Sensoren, Aktuatoren
- **Kommunikation**: Datenaustausch mit anderen weiter entfernten Geräten z.B. Netzwerk

8.2 Merkmale von I/O Geräten

- **Datenrate**: Volumen pro Zeiteinheit, das Übertragen werden
- **Anwendung**: Die Anwendung einens Gerätes kann variieren (z.B. verscheidene Zugriffsverhalten zwischen Dateiverwaltung und VMM)
- **Ansteuerungs-Komplexität**: unterscheidet sich auch zwischen den Geräten. Zum Beispiel durch einfache ansteuerung via Software, Memory Mapping und Busy Waiting bei Eingebetteten Systemen oder Ansteuerung durch Hardware wie DMA-Controller oder I/O Channels und Pufferung. (Pollen vs. Interrupt)
- **Transfereinheiten**: einzelne Zeichen vs. Datenblöcke
- **Daten-Repräsentation**: Byte Order, Parität usw.
- **Fehlerbehandlung**

8.3 I/O-Funktionen

- **Programmed I/O**:

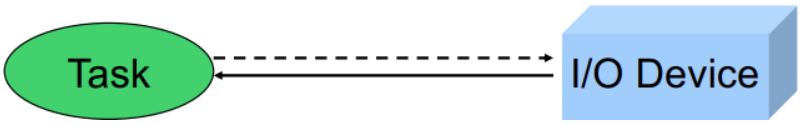

Abbildung 51: Programmed I/O

  - gesamte Logik in Software
  - Prozess schickt einen I/O Befehl und wartet mittels Busy Waiting

- **Interrupt-driven I/O**:

Abbildung 52: Interrupt-driven I/O

  - mehr Hardwareunterstützung als beim Programmed I/O
  - Prozess schickt einen I/O Befehl, Betriebssystem arbeitet währenddessen weiter
  - Betriebssystem wird durch einen Interrupt unterbrochen wenn das I/O Gerät fertig ist

- **Direct Memory Access (DMA)**:
  - mehr Hardwareunterstützung als beim Interrupt-driven I/O
  - Prozess schickt einen I/O Befehl
  - I/O Modul kopiert autonom Daten zwischen Speicher und I/O Modul
  - Betriebssystem wird durch einen Interrupt unterbrochen wenn das I/O Gerät fertig ist

Abbildung 53: Direct Memory Access

8.3.1 Direct Memory Access (DMA)

- Kann Aktionen der CPU nachbilden, indem DMA Controller die Kontrolle von Bus übernimmt.
- **effizient**: kein Kontextwechsel der CPU notwendig
- **flexibel**: ein DMA-Modul kann mehrere I/O Module versorgen (mehrere Ports oder eigener I/O Bus)
- es gibt zwei Arten wann ein DMA-Transfer startet:
  - blockweiser Transfer (eventuell parallel mit der CPU)
  - DMA erzwingt einzelne Übertragungszyklen am Bus (*cycle stealing*)

8.3.2 I/O Channel

- I/O Channel ist eine Erweiterung zum DMA
- kann im Gegensatz zu DMA I/O Protokolle ausführen
- Durch Ausführung von I/O Befehlen komplette Kontrolle über I/O Operationen
- I/O Instruktionen im Hauptspeicher, Abarbeitung durch I/O Prozessor, Interrupt Signalisiert das Ende
- Zwei Arten von I/O Channels:
  - **Selector Channel**: kann mehrere Geräte verwalten wobei immer nur eines aktiv ist
  - **Multiplexer Channel**: kann mehrere Geräte simultan verwalten

8.4 Kriterien für Betriebssytemdesign

Es muss ein Kompromiss zwischen zwei widersprüchlichen Zielen gefunden werden:

- **Effizienz**: wichtig das I/O Operationen oft das Bottleneck des Systems sind. Mehrere Prozesse bringt auch Swapping mit sich was auch zusätzliche I/O Operationen sind. → spricht für gerätespezifische Lösungen; jedes Gerät soll so angesteuert werden, dass die maximale Effizienz erreicht werden kann
- **Flexibilität**: für Einfachheit, geringere Fehlermöglichekeiten von I/O Operationen, Ersetzbarkeit und Austauschbarkeit der Geräte → einheitliche Schnittstellen erforderlich (z.B. Zugriffsfunktionen, wie `open`, `close`, `lock`, `unlock`, ...)

8.5 Logische Struktur von I/O

Um zwischen den beiden Zielen einen Mittelweg zu realisieren ist die logische Struktur von I/O in Schichten aufgebaut.

Bei diesem Model nimmt die Gerätespezifigkeit immer mehr zu je weiter *runter* es bei den Schichten geht.

8.5.1 I/O für Geräte mit Filesystem

Für Geräte mit Filesystemen führt man diese Schichtung auch noch nach oben fort. Als Programmierer soll man sich keine Gedanken machen wie eine Datei auf einer Festplatte abgespeichert ist. Dateien sollen geöffnet, geschlossen und bearbeitbar sein.

- Logical I/O besteht aus drei Komponenten

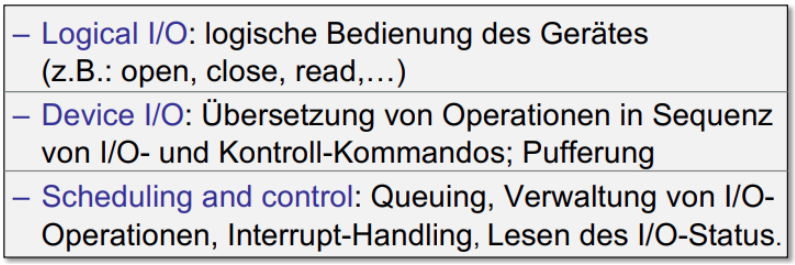

Abbildung 54: Schichtungsebenen

- **Directory Management**: Übersetzung symbolischer Dateinamen auf Dateireferenzen; Directory-Benutzeroperationen (z.B. add, delete, usw.)
- **File System**: behandelt die logische Struktur von Dateinen und Dateioperationen (z.B. `open`, `close`, `read`, `write`, usw.)
- **Physical Organisation**: Übersetung von logischen Referenze in Spuren und Sektoren auf der Platte.

8.6 Blocking vs. Non-Blocking I/O

- **Blocking**: Prozess wird von Zustand Running in die Blocked Queue gestellt und blockiert.
- **Non-Blocking**: I/O Operation wird sofort ohne blockieren fertiggestellt; return Status liefert Feedback über die Operation (z.B. Anzahl der übertragenen Bytes).

Benutzereingaben, bzw. wenn der Benutzer aufgefordert ist etwas einzugeben, werden beispielsweise mit blocking I/O realisisert. Auf der anderen Seite werden Computerspiele durch non-blocking I/O realisiert wo das Spiel zu einem bestimmten Tastendruck reagiert, aber wenn nichts eingegeben wird nicht blockiert sondern dennoch weiter läuft.

8.7 Synchrone vs. Asynchrone I/O

Weiters kann I/O in synchrone und asynchrone I/O unterteilt werden.

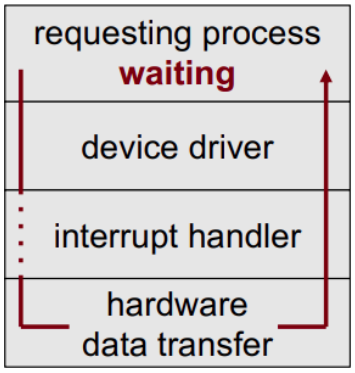

Abbildung 55: Synchrone I/O (blocking oder non-blocking)

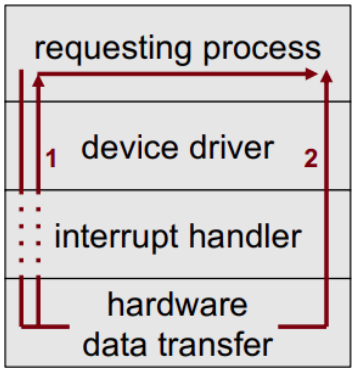

Abbildung 56: Asynchrone I/O

- **Synchrone I/O**: Der Prozess wartet auf den Abschluss der I/O. Das bedeutet nicht, dass er blockiert, oder Busy Waiting anwendet sonder der Prozess setzt erst fort, wenn die I/O Operation abgeschlossen ist.
- **Asynchrone I/O**: Die I/O Operation wird angestoßen, aber der Prozess setzt direkt mit der Abarbeitung seiner eigenen Instruktionen fort. Parallel zu dem Benutzerprozess wird die I/O Operation abgefertigt. Sobald die I/O Operation vollendet ist, wird mittels eines Interrupts dies dem Prozess signalisiert. Daraus lässt sich auch ableiten, dass Prozesse, wenn sie Asynchrone I/O Operationen verwenden möchten, passende Interrupt Handler brauchen. Außerdem muss der Prozess jederzeit die Rückmeldung der I/O Operation aufnehmen können → schwierige Logik für den Programmierer.

8.8 Synchroner I/O Request

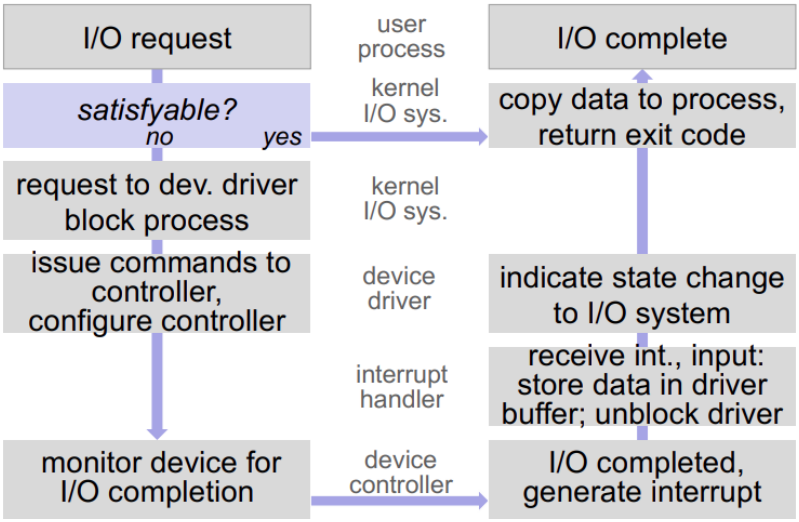

Abbildung 57: Synchroner I/O Request Zusammenspiel der Schichten

In diesem Beispiel wird ein Synchroner I/O Request aus einem Benutzerprozess dargestellt. Zuerst wird ein Mode Switch 2.10 durchgeführt. Die folgenden Schritte werden vom Betriebssystem abgearbeitet. Wenn eine Request direkt erfüllbar ist, können die angeforderten Daten sofort übermittelt werden. Wenn nicht, muss ein I/O Gerät angesprochen werden. Da dies meistens länger dauert, wird der Prozess währenddessen in die Blocked Queue verschoben. Der entsprechende Geräte Treiber wird angesprochen, der die tatsächliche Arbeit übernimmt. Als nächstes muss auf ein Interrupt Signal gewartet werden, welches signalisiert, dass die I/O Operation vollendet ist. Der Interrupt Handler speichert als nächstes die Daten in einem Puffer und der Geräte Treiber signalisiert dem Betriebssystem, dass die angeforderten Daten bereitgestellt wurden. Zuletzt müssen nur noch die Daten in den Prozessdatenbereich kopiert werden und ein Mode Switch erfolgen um die gesamte I/O Routine abzuschließen und den Prozess fortzuführen.
Dieser Prozess ist natürlich sehr langsam, vorallem wenn Daten Byte-weise aus einem Gerät ausgelesen werden müssen. Eine Lösung dazu ist das Puffern von I/O Anfragen.

8.9 Puffern von I/O Anfragen

Ein Puffer ist ein Zwischenspeicher für Daten bei einem I/O Transfer.

- Reduktion des Overheads: Zusammenfassung von low-level I/O Operationen (statt Byte-weises write wird der gesamte Puffer geschrieben oder beschrieben)
- Entkopplung von Prozess I/O und Betriebssystem I/O
- Im Betriebssystem Entkopplung von I/O und Swapping (Betriebssystem schreib in einen Puffer während ein Prozess ausgelagert ist; Prozess liest Puffer dann ein wenn er wieder anläuft)
- Abfangen von vorübergehenden Lastspitzen
- Geschwindigkeitsgewinn solange der Puffer nicht voll ist (Puffer zu beschreiben ist deutlich schneller als direkt mit dem I/O Modul zu kommunizieren)
- Geschwindigkeitsgewinn wird durch die Puffergröße begrenzt (bei Bursts)
- Langzeit-Performance durch Device Performance limitiert
- Nachteile:
  - Speicher muss reserviert werde
  - Puffermanagement: wer verwendet den Puffer? Wo bin ich im Puffer? usw.

Verschiedene Komplexitätsebenen des Pufferns sind in den folgenden Abbildungen graphisch illustriert. Die roten Boxen stellen den Speicher des Betriebssystems dar, die grünen den des Benutzer Prozesses. In allen Beispielen liest der Benutzerprozess Daten aus einem I/O Modul aus.

Abbildung 58: kein Puffer

Abbildung 59: einfacher Puffer

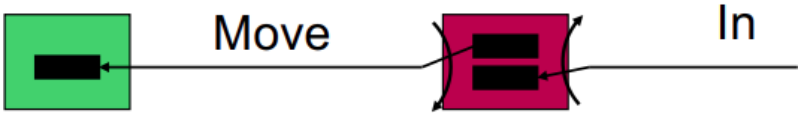

Abbildung 60: doppelter Puffer

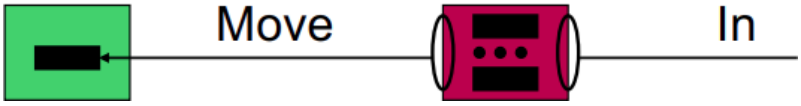

Abbildung 61: Ring-Puffer

8.10 Disk I/O Scheduling

In diesem Fall geht es um mechanische Magnetdatenträger.

- **Seek Time T_s**: mittlere benötigte Zeit, um Disk-Arm zur gewünschten Spur zu bewegen
- **Rotational Delay T_{RD}**: Zeitverzögerung, bis Anfang des gesuchten Sektors gefunden (im Durchschnitt eine halbe Umdrehung der Disk)
- **Transfer Time T_{TF}**: benötigte Zeit für die Übertragung der Daten
- **Average Access Time T_A**: mittlere benötigte Zeit für den Datenzugriff

8.11 Charakteristische I/O Zeiten

b...Anzahl der zu übertragenden Bytes
N...Anzahl der Bytes pro Spur
r...Umdrehungsgeschwindigkeit $[U/sec]$

$$T_{RD} = \frac{1}{2r}$$
$$T_{TF} = \frac{b}{r \cdot N}$$
$$T_A = T_S + T_{RD} + T_{TF}$$

Die Seek Time T_S ist noch unbekannt.

8.12 Strategien des Disk Scheduling

Das Ziel der verschiedenen Strategien ist die Minimierung der Seek Time T_S durch Ordnen der I/O Requests. Die Zeit die der Lese-Schreib-Kopf mit dem positionieren verbringt soll reduziert werden.

- **Priority**
- **First-In, First-Out (FIFO)**

- **Last-In, First-Out (LIFO)**
- **Shortest Service Time First (SSTF)**
  - die Request, wo sich der Lese-Schreib-Kopf am wenigsten bewegen muss wird als nächste Request ausgewählt
  - global gesehen sehr gut, da sich die mittlere Seek Time deutlich reduziert
  - Starvation ist allerdings möglich
- **SCAN** (Elevator Algorithmus)
  - auf dem Weg zu einem Request werden alle weiteren Request bedient denen *über den Weg gelaufen* wird
  - Lese-Schreib-Kopf fährt periodisch zur Disk-Mitte und zum Disk-Rand wie ein Aufzug.
  - ungleichmäßige Abarbeitung in der Mitte und am Rand
- **C-Scan**
  - im Prinzip genauso wie der SCAN Algorithmus
  - Nur noch eine *Fahrtrichtung* gefolgt von einer Leerfahrt
  - Unregelmäßigkeiten beseitigt
- **N-step-SCAN**
  - fasst n Request zu einem Bündel zusammen und arbeitet diese ab, unabhängig davon ob dazwischen neue hinzukommen
  - neue Request die genau im Weg *auftauchen* werden nicht mehr bevorzugt
- **FSCAN**
  - Hat zwei Gruppen von Request, ein aktive Gruppe wo alle Requests die momentan abgearbeitet werden drinnen sind und ein zweite in die neue Requests abgespeichert werden wenn die aktive Gruppe nicht leer ist. Sobald die aktive Gruppe abgearbeitet wurde, werden beide Gruppen vertauscht.

RAID (Redundant Array of Independent Disks): Beschleunigung und Fehlertoleranz
Diese Strategien können entweder vom Betriebssystem oder von der Festplatte selber realisiert werden → Prozessorentlastung

8.13 Disk Cache

Ein weiteres Prinzip für die Beschleunigung von I/O ist das Einsetzen von Disk Caches.

- Ähnliches Prinzip wie der Memory Cache
- Beruht auf das Lokalitätsprinzip 6.7.1
- Puffer im Hauptspeicher für Sektoren einer Disk
- Austauschstrategien
  - **Least Recently Used (LRU)**
  - **Least Frequently Used (LFU)**
  - **Frequency-Based Replacement**: Verbesserung von LFU
- Austausch im Cache bei Bedarf oder im Vorraus

8.13.1 Frequency-Based Replacement

Grundsätzlich werden die Cache-Blöcke nach ihrer Benutzungsfrequenz geordnet. Auf der linken Seite sind die Cache-Blöcke die am öftesten verwendet werden (Most Recently Used). Der zuletzt referenzierte Cache-Block ist immer ganz links zu finden. In einer reinen LRU Strategie würde immer der rechteste Block aus dem Cache entfernt werden. Bei der Frequency-Based Replacement Strategie wird mitgezählt wie oft ein Block referenziert wird. Ein Block startet seinen Lebenszyklus im roten Teil der *New Section*. Solange er in diesem Teil wieder referenziert wird wird der Zähler allerdings nicht erhöht. Erst wenn ein Block aus der *New Section* in die *Old Section* verdrängt wird und dann wieder referenziert wird, wir der Zähler inkrementiert. Wenn jetzt ein Block ausgetauscht werden soll wird die *Old Section* nach dem Block mit dem niedrigsten Zähler herausgesucht und entfernt.

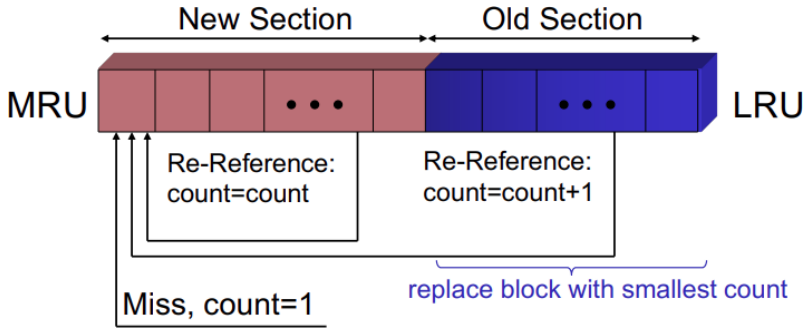

Abbildung 62: Variante 1

Diese Realisierung der Strategie hat allerdings einen grundlegenden Fehler. Jeder Block wird irgendwann einmal das erste mal in die *Old Section* rücken. Wenn dies passiert ist er der Block mit der geringsten Referenzanzahl. Wenn dann in diesem Augenblick ein Austausch stattfindet, wird der Prozess ausgewechselt der am am kürzesten von alles aus der *Old Section* im Cache war. Somit bekommt dieser Prozess gar nicht die Chance seinen Referenzzähler zu erhöhen.

Um dieses Problem zu beseitigen modifiziert man den Algorithmus etwas. Es wird ein neuer Teil eingeführt die *Middle Section*. In diesem Abschnitt werden die Blöcke nicht für die Ersetzung in betracht gezogen, können allerdings schon ihren Zähler durch Rereferenzierung erhöhen.

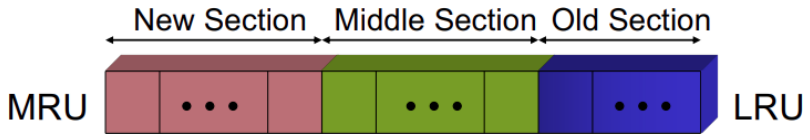

Abbildung 63: Variante 2

8.14 Zusammenfassung

- I/O ist ein Service des Betriebssystem zum Zugriff auf externe Geräte oder Ports des Prozessor
- Design von I/O ist ein Kompromiss zwischen Effizienz und Flexibilität
  - Schichtung der Software des I/O Systems
  - *Oben* einheitliche Befehle
  - *Unten* gerätespezifisch
- Puffern erhöht Performance und Flexibilität
  - Zusammenfassen von I/O Operationen
  - ganze Puffer bearbeiten
  - Puffer als Kopplung der der Prozess- und Systemaktivität
  - Burst können abgefangen werden
- Disk Scheduling und Caching zur Erhöhung der durchschnittlichen Zugriffsgeschwindigkeit

- Ordnen der Requests
- Caching als Performancegewinn

9 File Management

9.1 Motivation

- flüchtiger Arbeitsspeicher reicht nicht aus
- Daten sollen auch behalten werden wenn kein Strom da ist
- mehr Speicher als im Arbeitsspeicher vorhanden ist notwendig
- Dateien sollen über das Leben eines Prozesses hinaus verwendbar sein (Persistenz)

9.2 File Management

Eine Datei ist ein zentrales Element in vielen Prozessen und daher häufig verwendet. Es braucht also auch eine gewisse Organisation von diesen files. Ein Filesystem erfüllt genau diese Aufgabe und verwaltet files. Das sind einige Kernpunkte des File Managements.

- Namensgebung (Naming)
- Struktur
- Lokalisierung
- Zugriff
- Schutz (Protection)

Die Benutzersicht auf das Filemangement bezieht sich auf die Namensgebung der Dateien sowie das Verwalten, wo die Dateien in einer Verzeichnisstruktur abgelegt sind. Die Sicht des Betriebssystems auf das File Management beinhaltet den tatsächlichen Zugriffsschutz der Dateien sowie das abspeichern auf der Festplatte. Es spannt demnach eine Brücke zwischen der Benutzersicht und der physischen Repräsentation der Dateien.

9.3 File Management: Elemente

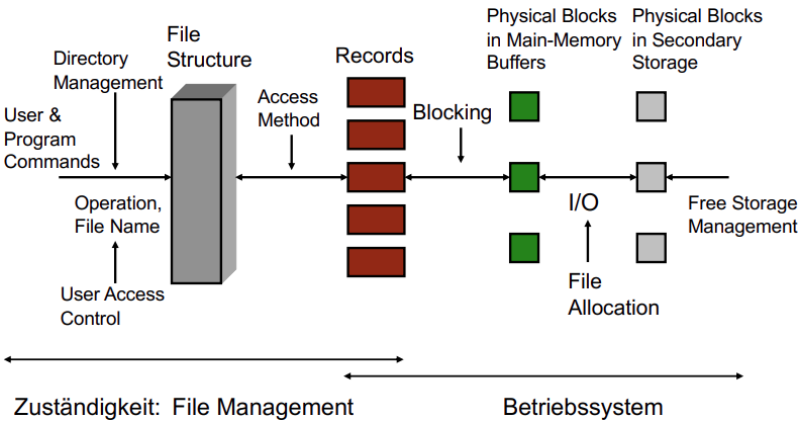

Abbildung 64: File Management: Elemente

In dieser Darstellung befindet sich die Benutzersicht auf der linken Hälfte. Dort sind die Fragestellungen für den Benutzer aufgelistet, wie beispielsweise das Finden und Benennen von Dateien. Dazu kommt das Navigieren im Dateisystem, ausführbare Dateien und Kommandos. All das möchte der Benutzer tun können.
Die roten Blöcke in der Mitte zeigen die interne Struktur der Dateien im System. Hier stellt sich die Frage wie Datein intern Strukturiert sind, ob diese bloß als Zeichenkette angesehen werden soll oder ob diese eine Record-Struktur haben (ähnlich wie bei Datenbanken wo Datensätze immer dieselbe Struktur haben). Darauf aufbauend stellen sich dann die Fragen wie schnell in diesen

Records navigiert und diese vernünftig organisiert werden können. Diese Fragen können entweder vom Betriebssystem oder von einem aufgesetzten Datenbanksystem gelöst werden.
In der letzten Ebene, ganz rechts, werden diese Records als Blöcke auf den Festplatten gespeichert. Hier ist eine zentrale Frage, wie diese Records, bzw. Zeichenketten auf diese Blöcke aufgeteilt werden und wie die Datenblöcke gefunden werden können die zu einer Datei gehören.
Die grünen Rechtecke dazwischen stellen den Austausch von Arbeitsspeicher und Sekundärspeicher dar. Hier geht es um Caching und Puffern der Daten, sowie das Laden von Blöcken.

9.4 File Management: Ziele

- kurze Zugriffszeit
- leichte Aktualisierbarkeit und Veränderbarkeit (löschen, hinzufügen und änder von Dateien)
- geringer Platzverbrauch, sparsamer Umgang mit dem Platz der auf der Festplatte gegeben ist
- gute Wartbarkeit
- Zuverlässigkeit

9.5 Datei-Organisation und Zugriff

Es stellt sich die Frage was eigentlich Dateien sind. Sind das einfach, wie heute oft von Betriebssystemen verstanden, eine Folge an Zeichen bzw. eine Folge von Bytes, oder ist das eine Strukturierte Sammlung also einer Sammlung an Records? Es gibt verschiedene Arten der Datei-Organisation:

- **unstructured sequence of bytes**
- **pile**: Records variabler Länge werden in Reihenfolge des Ankommens gespeichert
- **sequential file**: lauter Records mit fixem Format wobei ein Key-Feld die Position innerhalt der Datei bestimmt
- **indexed-sequential file**: Index als direkten Zugriff
- **direct (hashed) file**: Hashfunktion über Key-Feld, keine sequentielle Reihenfolge der Dateien

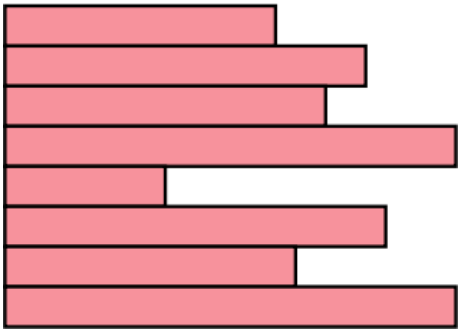

Abbildung 65: Pile

- Records variabler Länge
- variable Menge an Feldern
- chronologische Reihenfolge

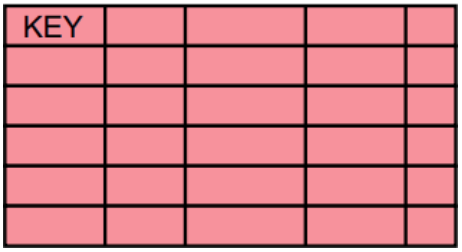

Abbildung 66: Sequential File

- Records fixer Länge
- fixe Menge an Feldern
- fixe Reihenfolge
- Reihenfolge ist durch einen Key bestimmt
- schneller lesender Zugriff

Um die Suche für den Key eines Sequentiellen Files zu beschleunigen wurden sich die Konzepte des Indexed Sequential File und Indexed File ausgedacht. Dazu kommt eine neue Datenstruktur der **Index** durch den die Schlüssel verwaltet werden. Durch diesen Index kann also direkt auf ein

bestimmtes file zugegriffen werden. Dadurch kann der Zugriff deutlich beschleunigt werden. Durch das Hinzufügen neuer files sowie das verändern bestehender files müssten eigentlich alle Keys angepasst werden. Da diese Operation sehr teuer ist werden diese Änderungen zuerst in ein Overflow file geschrieben und erst später realisiert. Das bedeute gelöschte files werden tatsächlich gelöscht, lücken gefüllt und neue files eingefügt. Zuletzt wird dann noch der Index neu generiert.
Wenn nach mehr als nur einem Schlüssel gesucht werden muss, müssen auch mehrere Index-Datenstrukturen verwendet werden.

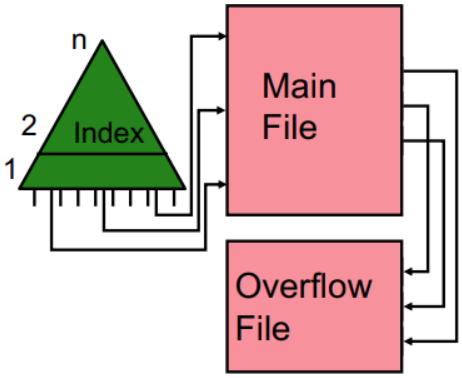

Abbildung 67: Indexed Sequential File

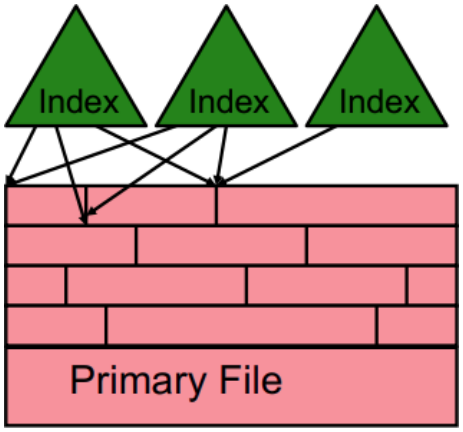

Abbildung 68: Indexed File

Eine Alternative zu Indexed Sequential File sind Hash Files:

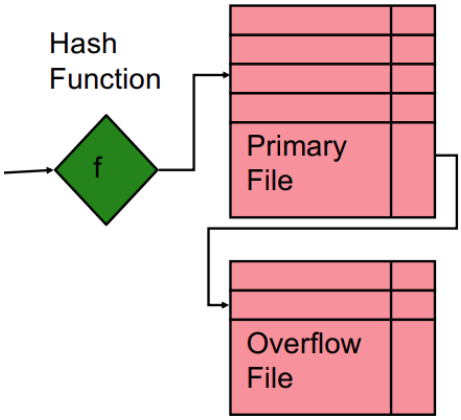

Abbildung 69: Hash File

Bei Hash Files wird statt dem Index eine Hashfunktion auf den Schlüssel angewendet der dann den Eintrag in einer Datei liefert, wo der gesuchte Datensatz zu finden ist. Wenn mehrere Datensätze auf denselben Eintrag mappen, werden diese in der Overflow Area abgelegt.
In Unix sowie Windows Systeme wird als Basisstruktur der Datei eine Sequenz an Bytes verwendet, die Strukturierung wird dann einem Datenbanksystem überlassen, welche auf diese Basisstruktur aufgesetzt wird.

9.6 File Types

- **Regular Files**: Nutzdaten die von einem Benutzer auf dem Computer gespeichert werden soll
  - **ASCII Files** vs. **binary Files**
  - **binary File**: Daten, Executables, usw., wobei das exe-File Format besonders hervorzuheben ist. Dieses File Format ist auch das einzige was ein Betriebssystem verstehen muss.

    * **Executable**: Muss einem bestimmten Format entsprechend aufgebaut sein, damit das Betriebssystem dieses interpretieren kann. Dieses Format ist in verschiedenen Segmente gegliedert:
        · Header
        · Text
        · Data
        · Relocation Bits
        · Symbol Table

      Außerdem kenntzeichnet eine sogenannte **Magic Number** im Header das exe-File

- **Directories**: spezielle Dateien die vom Betriebssystem unterstützt werden müssen, um im Dateisystem navigieren zu können, und dieses strukturieren zu können
- **Charcter Special Files**: Repräsentation sequentieller I/O Geräte welche Zeichenweise operieren
- **Block Special Files**: Repräsentation sequentieller I/O Geräte welche Blockweise operieren z.B. Festplatten

9.7 File Attributes

Sind für das Betriebssystem oder für die Unterstützung des Benutzers wesentlich.

- Creator, Owner, Protection, Password, Rechte
- **flags**:
  - Read-only flag
  - Hidden flag
  - System flag
  - Archive flag
  - ASCII/binary flag
  - Random Access flag
  - Temporary flag
  - Lock flags
- Creation Time, Time of last access, Time of last change
- Record length, Key position, Key length
- Current size, Maximum size

Welche Attribute es dann tatsächlich gibt hängt vom Betriebssystem ab.

9.8 File Names

Wie Dateinamen aufgebaut sind ist Sache der Designphilosophie des Betriebssystems. Da das Speichern der Dateinamen selber wieder Platz benötigt gab es in früheren Betriebssystemen einige Einschränkungen. Beispielsweise die Anzahl der Zeichen oder ob zwischen groß- und Kleinschreibung unterschieden wird. Als Beispiel sind die MS-DOS Betriebssysteme und Unix angeführt: Extensi-

	MS-DOS (Win95, Win98)	Unix
Dateinamen Länge	8 Zeichen + Extension	255 Zeichen
Relevanz von Groß- und Kleinschreibung	Nein	Ja
File Name Extensions	Teil des Namens	Konvention
Anzahl der Extensions	1	beliebig

ons werden bei Windows als seperate Eigenschaft angesehen und unterstützt während Unix diese als etwas vom Benutzer gewähltes ansieht und nicht genauer betrachtet.

9.9 File Operationen

- Create, Delete
- Open, Close
- Read, Write, Append
- Seek
- Get Attributes, Set Attributes
- Rename
- Lock

9.10 Dateiverzeichnis

- Verzeichnis gespeicherter Daten
- Liefert Abbildung von Datenamen auf Daten
- Einträge: Name, Attribute, physische Adresse der Daten
- Struktur:
  - einfache Liste in anfänglichen Betriebssystemen
  - hierarchische Baumstruktur von einem Root Directory ausgehend
- Verzeichnisse in Hash-Struktur gespeichert für schnellere Zugriffszeiten
- in einem Verzeichnis werden Verweise zu anderen Dateien abgespeichert

9.10.1 Verzeichnis mit Baumstruktur

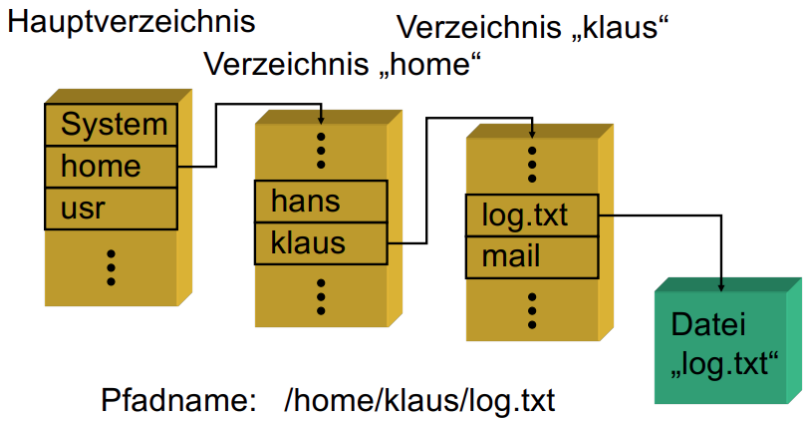

Abbildung 70: Verzeichnis mit Baumstruktur Beispiel

Im Hauptverzeichnis finden sich normalerweise einige vordefinierte Verzeichnisse für das Betriebssystem. Beispielsweise wo Programmdateien des Betriebssystem abgelegt sind oder Verzeichnisse für Benutzer und Systemdaten. Üblicherweise verweist das *Home* Directory auf die Benutzerdaten.

9.11 Pfadnamen

Ermöglicht das eindeutige identifizieren einer Datei in einem Dateisystem. Dabei gibt es zwei Arten von Pfadnamen:

- **absolut**: identifiziert Datei durch Beschreibung des Pfads von der *Root* ausgehend.
  - Windows: `\usr\hans\mailbox`
  - Unix: `/usr/hans/mailbox`

- **relativ**: lokalisiert Datei vom **Working Directory** (derzeitiges Verzeichnis) aus. Wenn beispielsweise das Working Directory `/usr` ist, so kann auf dieselbe Datei wie im vorherigen Beispiel mit `hans/mailbox` zugegriffen werden.
- **Current Directory**: . (dot)
- **Parent Directory**: .. (dotdot)

9.12 Directory Operationen

- Create, Delete
- Opendir, Closedir
- Readdir
- Rename
- Link, Unlink (Verweise werden eingefügt bzw. herausgenommen)
- Änderung der Zugriffsrechte

9.13 File System Implementierung

Die Implementierung des File System beschäftigt sich mit den Aufgaben des Betriebssystems. Es geht darum wie Files und Directories tatsächlich gespeichert werden, wie der vorhanden Platz auf der Festplatte verwaltet wird und wie Zuverlässigkeit und Performance erreicht werden können.

9.13.1 Beispiel: File System Layout

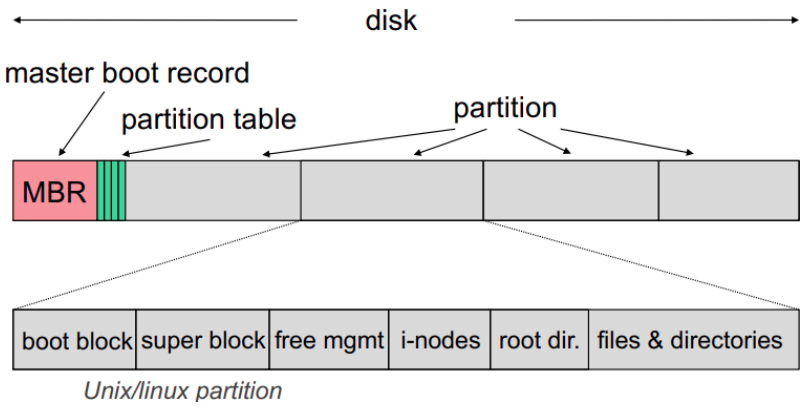

Abbildung 71: File System Layout Beispiel

Dieses Diagramm zeigt das typische Layout eines Dateisystems auf einer Festplatte. Es gibt einen sogenannten **Master Boot Record** (MBR) welches der erste Block auf der Festplatte ist. Danach gefolgt kommt eine Partitionstabelle gefolgt von den Partitionen. Jede **Partition** ist ein alleinstehendes vollständiges Dateisystem. Beim Start des Computers wird das MBR ausgelesen und interpretiert. Dieses liest dann die Partitionstabelle aus und wählt dann eine Default-Partition, bzw. befragt den Benutzer, aus um das darauf gespeicherte Betriebssystem zu starten. Das geschieht indem das Boot-Record auf dieser Partition ausgelesen und ausgeführt wird.

- Disk Unterteilung in Partitionen mit unabhängigem Filesystem
- Master Boot Record (MBR) im Sektor 0 der Disk
  - Boot Code
  - Partition Table (start, end of partitions, active partition)
- Systemstart: BIOS exekutiert Code des MBR
  - lokalisieren der aktiven Partition
  - Ausführen des ersten Blocks (Boot Block) → Laden des Betriebssystems der aktiven Partition

9.13.2 Datei Implementierung

- Eine Datei besteht im Sekundärspeicher als Sammlung von Blöcken.
- Wo sind die Blöcke eines Files zu finden?
- Verschiedene Strategien der Block-Allokierung
  - Contiguous Allocation
  - Chained Allocation
  - Indexed Allocation
  - I-Nodes

Contiguous Allocation eine Datei belegt eine einzige, aneinander grenzende Menge von Blöcken

- **Vorteil**:
  - gute Performance beim Lesen
- **Nachteil**:
  - Platzprobleme beim Vergrößern einer Datei
  - externe Fragmentierung
- **Verwendung**:
  - CD-ROMs
  - DVD-ROMs

Chained Allocation Belegung einzelner Blöcke, die über Zeiger auf den Folgeblock verkettet sind

- **Vorteil**:
  - keine externe Fragmentierung
- **Nachteil**:
  - keine Lokalität der Blöcke
  - langsamer Zugriff bei Random Access (Wenn eine bestimmte Stelle einer Datei gesucht wird muss die gesamte Pointer-Kette durchlaufen werden)
  - Nutzdaten pro Block $< 2^n$, da der Zeiger auf den Nachfolgeblock selbst Platz im Block braucht

Indexed Allocation wie Chained Allocation, allerdings werden die Pointer in einer Tabelle dem File Allocation Table (FAT) im Speicher und nicht in den Blöcken der Datei gehalten

- **Vorteil**:
  - Sowohl direkter als auch sequentieller Zugriff gut unterstützt
  - Blöcke ganz für Nutzdaten verfügbar
- **Nachteil**:
  - großer Platzbedarf für FAT im Arbeitsspeicher

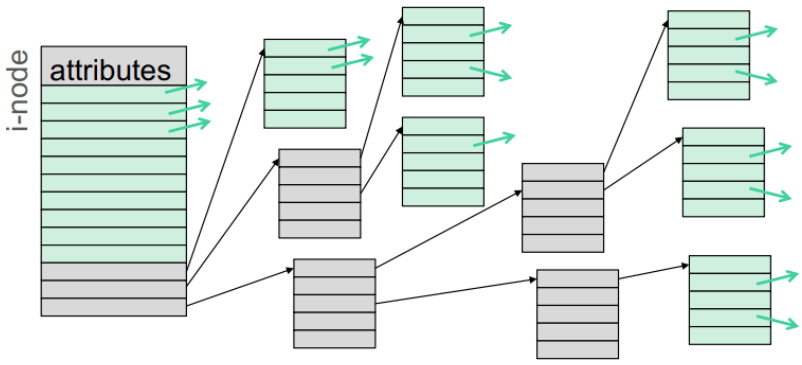

Abbildung 72: I-Node Beispiel

I-Nodes Datenstrukur für jedes File, enthält Fileattribute und Referenzen auf die Blöcke des Files.

- **Vorteil**:
  - I-Node wird nur im Speicher gebraucht, wenn ein File verwendet wird (dafür notwendig ist ein Array, das I-Nodes für maximale Anzahl offener Dateien halten kann)
  - Blöcke ganz für Nutzdaten verfügbar
- **Nachteil**:
  - Anzahl der Blockreferenzen pro I-Node ist begrenzt → Verwendung indirekter, doppelt und dreifach indirekter Blöcke

Die doppelten oder dreifachen Verweise kommen allerdings in der Praxis nicht so oft vor, da in etwa 95% der Dateien nur eine Größe von 2-4 Kilobyte aufweisen - genug um in einem I-Node erfasst zu werden. Erst bei seltenen größeren Dateien werden dann multiple Referenzen benötigt.

9.13.3 Sequential Files - Blocking

Es gibt verschiedene Methoden zur Abbildung von Records auf Blöcke:

- Fixed Blocking
- Variable-length spanned Blocking
- Variable-length unspanned Blocking

Fixed Blocking Records fixer Länge, integrale Anzahl von Records pro Block → Verschnitt

Abbildung 73: Fixed Blocking Beispiel

Variable-length spanned Blocking Records variabler Länge, Records können auch auf zwei Blöcke verteilt sein → kein Verschnitt

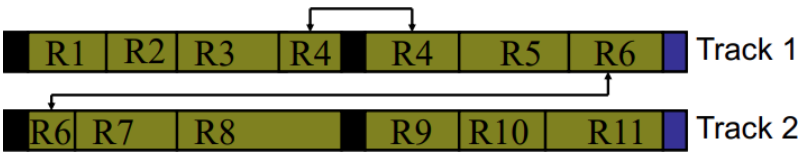

Abbildung 74: Variable-length spanned blocking Beispiel

Abbildung 75: Variable-length unspanned blocking Beispiel

Variable-length unspanned Blocking Records variabler Länge, Record als ganzes in einem Block gespeichert → Verschnitt

9.14 Root-Directory Implementierung

Auffinden von Dateien:

1. Lokalisierung des Root Directories
2. Interpretation es Pfadnamens (Verzeichnis für Verzeichnis)

Position es Root Direktories

- fixe Position vom Partitionsanfang aus
- Unix: Startadresse der I-Nodes im Super Block erster I-Node verweist auf das Root Direktory
- Win: Boot Sector enthält Informationen über die Adresse der Master File Table (MFT)

9.14.1 Directories und File Attribute

Wo findet das Betriebssystem die File Attribute?

- **File Attribute in Directory-Einträgen**: Beispielsweise Directory Einträge fixer Größe der Form: Filename (fixe Größe), Struktur mit File Attributen, eine oder mehrere Blockadressen. Dieses Prinzip wird von Windows verfolgt.

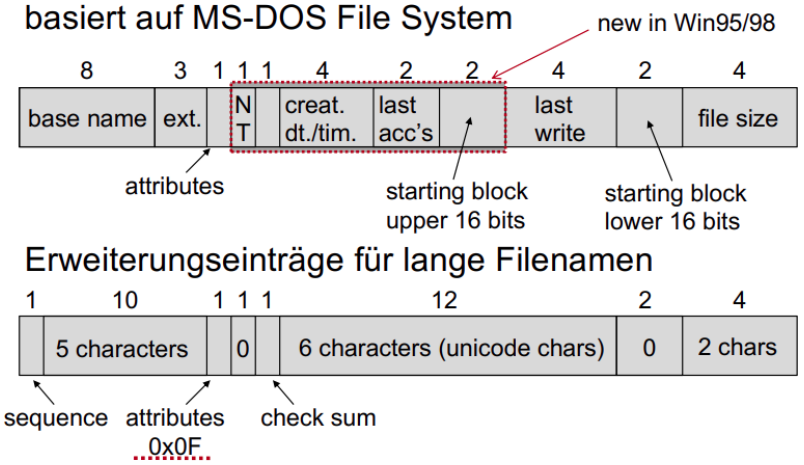

Abbildung 76: Directory Einträge in Win95 und Win98

- **Ablegen der Attribute in I-Nodes**: Directory Einträge der Form: Filename, I-Node Nummer. Dieses Prinzip wird von Unix verfolgt.

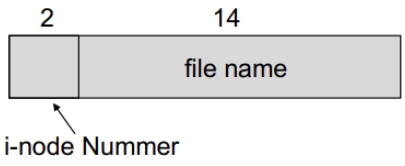

Abbildung 77: Directoriy Einträge in Unix

9.15 File Block Size

Je größer die Block Size ist, desto höher ist auch die Datenrate, da ganze Blöcke auf einmal bearbeitet werden können. Allerdings steigt dann auch die Chance, dass zu kleine Dateien einen Datenblock nicht mehr vollständig ausfüllen, was zu einer geringeren Nutzung des verfügbaren Speichers führt.

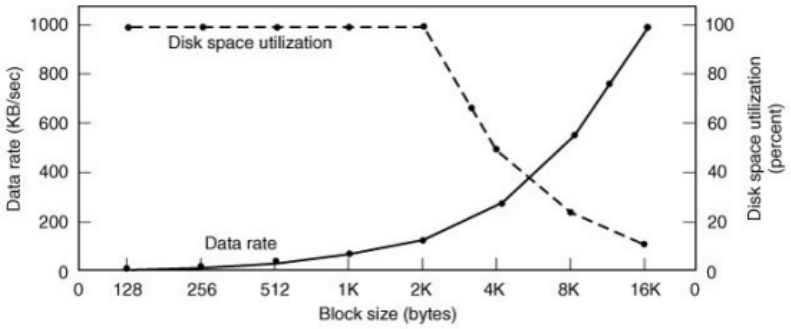

Abbildung 78: Datenrate und Nutzung des Speicherplatzes in Verhältnis zur Block Size

9.16 Verwaltung freier Blöcke

Disk Allocation Table zur Markierung freier Blöcke.

- **Chained Free Portions**: Alle freien Bereiche verbunden per Zeiger-Eintrag. Problem: mit der Zeit kommt es immer mehr zur Fragmentierung → Overhead für Zeigerupdate (=R/W) bei File Operationen. Erzeugt eine Kette an freien Blöcken.
- **Bit Tables**: Bitvektor mit je einem Bit pro Plattenblock. Geringer Platzbedarf und bietet guten Überblick über Folgen von freien Blöcken.
- **Indexing**: Freie Blöcke als eigenes File betrachtet. Effizient für alle Datei-Belegungsverfahren.

9.17 Performance

- **Disk Caching**:
  - Ausnutzen der Lokalität (Lokalitätsprinzip 6.7.1)
  - Bedeutung eines Blocks für die Konsistenz des File Systems → Write Back
- **Block Read Ahead**: Spekulation, dass wenn ein Block gebraucht wird, die nächsten Blöcke auch gebraucht werden.
- **Kopfbewegungen der Disk**: Minimieren der Kopfbewegung durch Positionen der Daten auf der Disk zur Steigerung der Performance

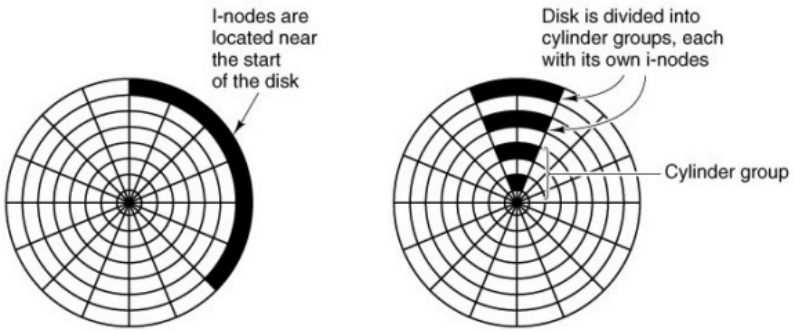

Abbildung 79: Position der Daten auf der Disk

Wenn die I-Nodes näher an den tatsächlichen Daten sind kann sich Bewegung gespart werden.

9.18 Zusammenfassung

- File ist zentrales Element der meisten Programme
- Persistenter Speicher notwendig
- User-Sicht vs. Designer-Sicht
  - Benutzersicht: Zugriffsrechte; Was dürfen Programme?
  - Betriebssystemsicht: Brücke zwischen den Anforderungen des Benutzers und der Organisation auf der Festplatte
- Filesystemstruktur
  - Wie werden Dateien organisiert?
  - Wie werden Dateien gefunden?
- Files, Directories, Special Files
- Abbildung von Files auf Datei Blöcke, Verwaltung freier Blöcke

10 Security

Security umfasst alle Strategien, Vorkehrungen und Tools um die Vertraulichkeit, Integrität und Verfügbarkeit von Informationen und Dienste einer Organisation zu gewährleisten. Um Sicherheit gewährleisten zu können spielen einige Aspekte mit:

- **Bedrohungsszenarien**: Was gilt überhaupt als sicher? Wovor muss ich mich schützen?
- **Abläufe und Strategien**: Was muss ich tun um Informationen sicher zu halten? Wer hat Zutritt zu welchen Informationen?
- **Computerspezifische Fragen**: Wie schaut mein Sicherheitskonzept aus und wie wird dieses von dem Computer oder Computersystem unterstützt?
- **Basis**: Vertrauen, Annahmen über Vertrauenswürdigkeit; Durch das installieren von Sicherheitsvorkehrungen werden Werkzeuge und Komponenten anderer Menschen oder Firmen benötigt. Es wird darauf Vertraut, dass diese Personen keine Informationen darüber Verbreiten oder das das Wissen über diese Sicherheitssysteme ausnutzen. (z.B. dass der Schlosser nicht einen zweiten Schlüssel anfertigt und dann die Wohnung ausräumt)

In der Praxis wird es nicht möglich sein sich vor allem zu schützen da für jede Schutzmaßnahme Kosten und Mühen ansteigen. Aufgrund dessen werden oft Abstriche bei Sicherheitsvorkehrungen gemacht, welche nicht immer notwendig sind, bzw. Sicherheitsvorkehrungen sehr spezifisch auf den Einsatz zugeschnitten. Beispielsweise werden private Rechner anders geschützt als Firmen interne Computersysteme, welche vertrauliche Dokumente beherbergen.

10.1 Security: Objectives (CIA-Triad)

- **Confidentiality**: Geheimhaltung; Wer darf Daten sehen?
- **Integrity**: Daten entsprechen dem Sollzustand; Wer darf Daten ändern?
- **Availability**: Verfügbarkeit der Daten, wenn diese verfügbar sein sollen.

10.1.1 Security Concerns

Diese zwei Ziele können als Subziele zu den drei davor angeführten Zielen hinzugefügt werden:

- **Authenticity**: Korrektheit der Identität und berechtigter Zutritt
- **Accountability**: Nachvollziehbarkeit, Protokollierung von Zugriffen; Wer hat was gemacht?

10.2 Types of Security Threats

- **Passive Threats**
  - Abhören oder Monitoring von Informationen ohne Wissen des Betroffenen
  - Confidentiality Violation
- **Active Threats**
  - Aktive Manipulation von System und Daten
  - meist schwerwiegender als Passive Threats
  - Beeinträchtigung von Integrität und Service

10.2.1 Security Threads

- **Denial of Service** (Interruption)
  - Vorübergehende oder permanente Unterbrechung eines Services
  - physikalische Zerstörung, Überlastung, usw.
  - zielt auf die Availability ab

Abbildung 80: Denial of Service

- **Exposure / Interception**
  - nicht autorisierter Lesezugriff

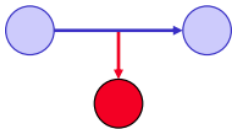

Abbildung 81: Exposure / Interception

- **Modification / Fabrication**
  - Verletzung der Datenintegrität durch Änderung oder Generierung von Daten
  - Theft of Service: unautorisierte Verwendung von Ressourcen (z.B. Rechenzeit)

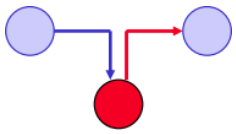

Abbildung 82: Modification / Fabrication

Goals vs. Threats

10.3 Intrusion

- **Ziele**
  - Verschaffen eines Zugangs zu einem System
  - Erhöhung der Privilegien
- **Methoden**
  - Ausnützung einer Sicherheitslücke
  - Aneignung eienes Passworts

Goal	Threat
Confidentiality	Exposure / Interception
Integrity	Modification / Fabrication
Availability	Denial of Service

10.3.1 Intruders (Adversaries)

- Gelegenheitsattacken von nicht versierten Benutzern
- gelegentliche Attacke von technischen Insidern
- gezielte Versuche der Bereicherung
- Industrie- und Militärspionage

10.4 Maleware

- **Virus**: In einem Programm versteckter Code, der sich selbst in andere Programme kopiert
  - eventuell zusätzlich destruktives Verhalten
  - Denial Of Service (DoS) durch Ressourcenblockierung
  - Distributed Denial Of Service (DDoS) Attacken
  - Key Logger
- **Worm**: Programm, das sich selbst repliziert und Kopien seines Codes über ein Netz an andere Computer sendet (z.B. über mail oder remote login)
  - Bsp. Windows: kopiert sich in User *Startup Dir.*
- **Trojan Horse**: Programm mit gewünschter Funktionalität, das versteckten Code mit bösartiger Funktionalität enthält
  - oft vom Benutzer selbst erworben (z.B. Download)
  - Schutz nur durch Vertraulichkeit gegeben
- **Logic Bomb**: Programmstück, das sich selbst bei Auftreten einer Bedingung aktiviert
  - z.B. Zeitablauf: kein Login eines Mitarbeiters für eine gewisse Dauer → Start von bösartigem Code
  - benötigt Zugriff auf das System für die Installation
- **Trapdoor**: Geheimer Einstiegspunkt, Umgehung der Zugriffskontrolle
  - benötigt Zugriff auf das System für die Installation
- **Port Scan**: automatisierter Versuch des Verbindungaufbaus zu unterschiedlichen Ports, um bekannte Fehler auszunutzen
- **Denial of Service (DoS)**: Überlastung eines Rechners, sodass dieser kein sinnvolles Service mehr zur Verfügung stellen kann
- **Distributed Denial of Service (DDoS)**: gleichzeitige DoS Attacke von unterschiedlichen Rechnern

10.5 Typische Methoden von Attacken

- Anfordern und Auslesen von Speicher (Memory, Disk, ...) → Betriebssystem muss sicherstellen, dass von Prozessen allokierter Speicher keine sensiblen Daten beinhaltet
- Aufruf unerlaubter System Calls bzw. erlaubter Calls mit unerlaubten oder *sinnlosen* Parametern (Kontrolle durch Absturz des Systems)
- Verwendung von Abbruchtasten (`DEL`, `BREAK`, etc.) während des Loginprozesses (Zwingen der Software in einen Ausnahmemodus)

- Modifikation von Betriebssystem-Datenstrukturen, die im User Space gehalten werden
- Fälschung der Login-Routine, Abhören unverschlüsselter Verbindung (zB. über telnet)
- Wenn das Manual sagt: „Do not do X“, möglichst viele Varianten von X ausprobieren
- Bitte an Systemprogrammierer, bestimmte Sicherheitsüberprüfungen für eigenen Account auszuschalten
- Social Engineering (z.B. Phishing, Bestechung)

10.5.1 Beispiel Stack/Buffer Overflow Attack

- Über Pufferende schreiben und damit Stack verändern (möglich wenn der Puffer Overflow nicht geprüft wird)
- Exploit Code am Stack positionieren
- Returnadresse am Stack auf Exploit-Code setzen

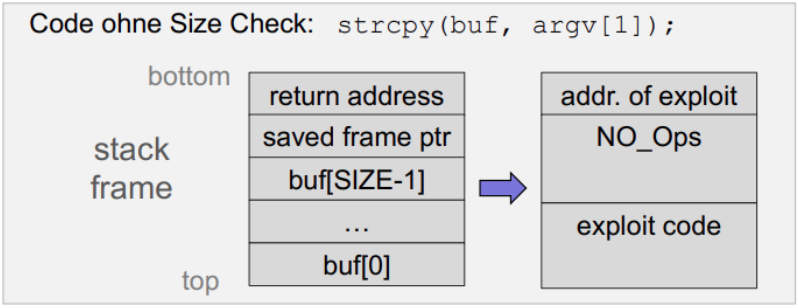

Abbildung 83: Stack/Buffer Overflow Attack

10.6 Design Priniciples for Security

- **Open Design** und nicht „Security by Obscurity“; Verlassen auf Meldung von etwaigen Fehlern in der Software der Community → System wird über die Zeit immer sicherer
- **Default-Einstellung**: keine Berechtigung
- **Least Privilege**: Jeder Benutzer sollte immer die minimalsten Rechte bekommen, die er für seine Aufgabe benötigt
- **Economy of Mechanisms**: Einfachheit der Sicherheitsmechanismen, Implementierung auf möglichst niedriger Ebene; besser wenige simple Sicherheitmechanismen Anwenden, als viel komplexe, um die Übersicht zu gewährleisten. Sonst erhöht sich die Chance auf eine unentdecktes Schlupfloch.
- **Acceptability**: Es sollten nur Sicherheitsmechanismen verwendet werden die akzeptierbar sind. Beispielsweise stellt sich immer wieder die Frage ob der *Passwortmechanismus* akzeptierbar ist, da heutzutage unmengen an Konten angelegt werden und man sich nunmal nicht für jedes Konto ein eigenes Passwort merken kann → Benutzer verwenden gleiche Passwörter oder immer leichtere Passwörter, da diese leichter zu merken sind.
- **Überprüfung gegenwärtiger Berechtigungen**: Routinemäßiges periodisches Überprüfen der Berechtigungen, nicht nur beispielsweise nach dem Login die Berechtigungen setzten.
- **Complete Mediation**: Kontrolle aller Zugriffe auf Ressourcen (auch Ausnahmesituationen); Wartungsmodus muss auch gesichert sein.

10.7 User Authentification

Der Begriff Authentification fasst das Vertrauen, dass ein Benutzer oder Prozess der ist, der er vorgibt zu sein, zusammen. Das kann durch verschiedene Arten überprüft werden:

- **Besitz eines Schlüssels**: physikalischer Schlüssel, Chipkarte, usw.
- **Überprüfung von Attributen**: Fingerabdruck, Iris; Allerdings sind diese Methoden sehr aufwändig und teuer und haben Probleme bei der Akzeptanz der breiten Masse.
- **Abfrage von Wissen**: Passwort ist der gängigste Mechanismus. Ein Vorteil von Wissensabfragen ist, dass sie sehr leicht zu implementieren sind. Nachteil des Passwortmechanismus beispielsweise ist die Qualität der Passwörter die ja vom Benutzer gewählt werden.

10.8 Passwörter

Das Problem bei Passwörtern, ist ein Passwort zu wählen, das sicher ist. In diesem Fall bedeutet Sicherheit, dass das Passwort nicht herausgefunden werden kann, bzw. dass es den Standard-Passwortattacken standhält.

10.8.1 Suchraum

Anzahl an Passwörtern mit eine Länge von 7 Zeichen:

$$95^7 \approx 7 \cdot 10^{13}$$

Im Vergleich wäre die Anzahl an Passwörtern mit einer Länge von nur 6 Zeichen und nur Klein- oder Großbuchstaben:

$$26^6 \approx 3 \cdot 10^8$$

Eine Anzahl, die durchaus von einer *Brute Force Attack* leicht verwertet werden kann. Um solche Attacken schwieriger zu machen werden Passwörter verschlüsselt abgespeichert.
Außerdem versucht man die Suche auch noch durch sogenannte *Salt Bits* zu erweitern. Das sind n sichtbare Bits, die bei der Verschlüsselung mitverwendet werden. Dadurch vergrößert sich der Suchraum (und damit auch der Rechenaufand) um 2^n. Unix verwendet beispielsweise ein 12 bit salt.

10.8.2 Standard-Passwortattacken

- Default Passwort
- Durchprobieren aller kurzen Passwörter (z.B. alle Buchstabenkombinationen mit 6 Zeichen)
- Verwendung von Wörterbüchern und Namen
- Wörterbuch und typische Ersetzungen oder Änderungen wie zum Beispiel „i“→ 1 oder „o“→ 0. Groß- Kleinschreibung wechseln, Umkehrung usw.
- Benutzer-spezifische Informationen (z.B. Geburtsdatum, Haustier, KFZ-Kennzeichen, usw.)

Ein *Standard Password Cracker* kann und bedient sich an den ersten vier hier aufgelisteten Punkten.

10.8.3 Passwortattacken: Maßnahmen

- One-Time Password; Transaktionsnummern (TAN) wird häufig im Bankwesen verwendet
- Zeitablauf von Passwörtern
- Challenge Response Protokolle
- Zeitverzögerung Logins (vor allem nach einer gewissen Anzahl an Fehlversuchen)
- Logging, Anzeige der Daten des letzten Logins

10.9 Protection

Sicherstellen des kontrollierten Zugangs zu Programmen und Daten auf einem Computersystem im Vergleich zum Zugang eines Profils.

10.9.1 Protection Domains

- **Objekte**: CPU, Speichersegmente, Files, usw.
- **Zugriffsrecht**: Paar (Object, Recht)
- **Domain**: Menge von Zugriffsrechten

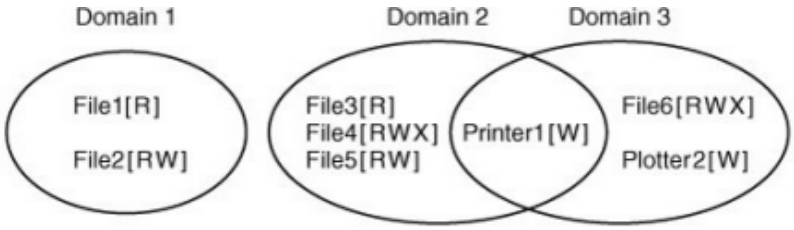

Abbildung 84: Protection Domains

In Unix sind beispielsweise solche Domains durch User- bzw. Group-IDs definiert (UID, GID). Die Protection Domain kann sich fortlaufend ändern.

- User Part vs. Kernel Part eines Prozesses
- `exec` von File mit gesetztem `SETUID`, `SETGID` Bit

10.9.2 Access Matrix

Ein Prozess mit der Domain D_i darf eine Operation *op* auf einem Object O_j ausführen wenn *op* in der Access Matrix angegeben ist.

	Object							
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Abbildung 85: Access Matrix

Dieses Modell ist statisch und erlaubt so noch kein Änderungen an Rechten. Allerdings braucht ein System normalerweise die Möglichkeit Berechtigungen auch zu modifizieren.

- Operationen zum Hinzufügen und Löschen von Rechten
- Spezielle Zugriffsrechte
  - Owner: spezielle Rechte des Besitzers
  - Kopieren, Verändern von Rechten
  - Transfer: Domainwechsel des Prozesses

	Object										
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Abbildung 86: Access Matrix mit Dynamic Protection

Auf der rechten Seite dieser Tabelle ist auszulesen, dass die Domain 1 in die Domain 2 wechseln darf.

Mechanismus vs. Policy

- **Mechanismus**
  - Betriebssystem stellt Access Matrix und Regeln zur Verfügung
  - Stellt Einhaltung der Regeln sicher
- **Policy**
  - Benutzer bestimmen Policy
  - Wer kann welches Objekt wie ändern?

10.9.3 Access Control List (ACL)

Access Matrizen haben sehr viele leere Felder. Um das zu umgehen gibt es mehrere Ansätze, bei einer davon stellt man sogenannte Access Control Lists auf. Bei diesen Access Control Lists sind die Zugriffsrechte bei den Objekten gespeichert, also so gesehen eine Spaltenzerlegung der Matrix.

- Domain: Differenzierung nach Benutzergruppen
- leichtes Ändern der Zugriffsrechte von Objekten

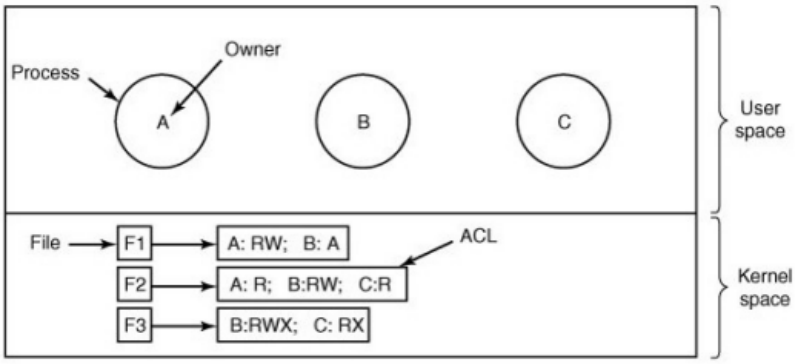

Abbildung 87: Access Control List

10.9.4 Capability Lists

Mit dieser Strategie möchte man auch die leeren Felder einer Access Matrix umgehen. Für jeden Prozess gibt es eine List mit erlaubten Zugriffsoperationen.

- Sogenannte Capability Tickets regeln Zugriff auf Objekte durch Ticket-Besitzer
- Weitergabe und Vererbung von Tickets
- Fälschungssicherheit von Tickets wird benötigt → Verschlüsselung

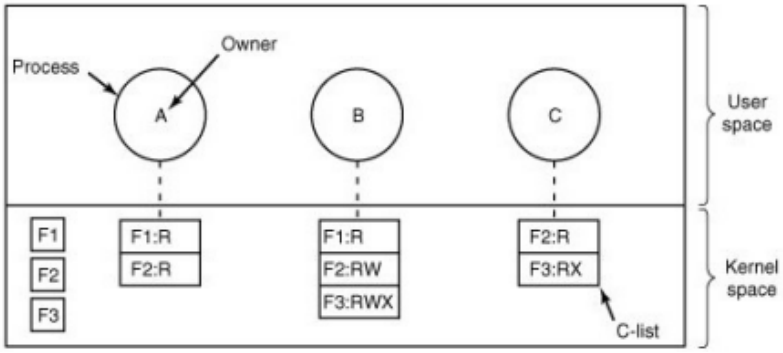

Abbildung 88: Capability Lists

10.9.5 Lock-Key System

Auch bei dem Lock-Key System wird keine zugrundeliegende Access Matrix verwendet, anstelle dessen werden Locks und Keys an Objekte bzw. Domainen vergeben.

- jedes Objekt hat eine Menge von Locks (eindeutiges Bitmuster)
- jede Domain besitzt eine Menge von Keys
- Ein Prozess in der Domain D_j darf auf das Objekt O_i zugreifen, wenn ein Key von D_j zu einem Lock von O_i passt

10.9.6 Bell und LaPadula's Model

- Regeln für den Informationsfluss
- Hierarchie von Security Classifications (SC) für Subjects und Objects
  - top secret
  - secret
  - public
- Operationen von Subjects auf Objects:
  - read-only (keine Modifikationen)
  - append (kein lesen)
  - execute (kein lesen oder schreiben)
  - read-write

Security Axioms

- **Simple Security Property**
  - **Read**: „no read up", für ein Subject S (z.B. ein Prozess) und einem Objekt O muss gelten
    $$SC(S) \geq SC(O)$$
- **The *properity** (star property)
  - **Append**: „no write down"meint, dass keine vertrauliche Information aus höheren Kreisen in niedere Kreise *geleaket* werden soll.
    $$SC(S) \leq SC(O)$$
  - **Read and Write**:
    $$SC(S) = SC(O)$$

Ein Subject S kann ein Objekt O_1 lesen und ein zweites Objekt O_2 schreiben, wenn gilt:

$$SC(O_1) \leq SC(S) \leq SC(O_2)$$

10.10 Kryptographie als Security Tool

- **Geschlossenes System**: Betriebssystem kann Sender und Empfänger von IPC kontrollieren. → keine Kryptographie notwendig
- **Offenes System**:
  - Ohne Kryptographie kein Vertrauen auf Korrektheit von Sender und Empfänger von Nachrichten.
  - Kryptographie beschränkt potentielle Sender und Empfänger von Nachrichten und deckt auch Veränderung bzw. Manipulation ab. Schlüssel wird für das Lesen und Schreiben benötigt.
- Basiert auf dem Besitz geheimer Schlüssel
- Verschlüsselung, Authentifizierung

10.11 Intrusion Detection

- **Threshold detection** z.B. viele Loginversuche
- **Anomaly detection**: Regeln zum Aufspüren von Anomalien. Damit man Anomalien überhaupt erkennen kann müssen auch reguläre Aktivitäten Protokolliert werden. Das geschieht über ein sogenanntes Audit Record.
- **Audit Records**: Aufzeichnung über Operationen
  - Subject
  - Action
  - Object
  - Exception Conditions
  - Resource Usage
  - Timestamps

Zusammen mit dem Audit Record und dem typischen Verhalten in einem System können dann Anomalien herausgefunden werden. Allerdings ist nicht jede Anomalie gleich eine schlechte. Beispielsweise werden Passwörter nur selten geändert und wenn dies einmal auftritt wird es sehr wahrscheinlich als eine Anomalie markiert werden. Dennoch sind Passwortänderungen eigentlich etwas gewolltes und nichts schädliches. → Systemadministrator muss entscheiden welche Anomalie bösartig ist.

10.12 Security Architecture

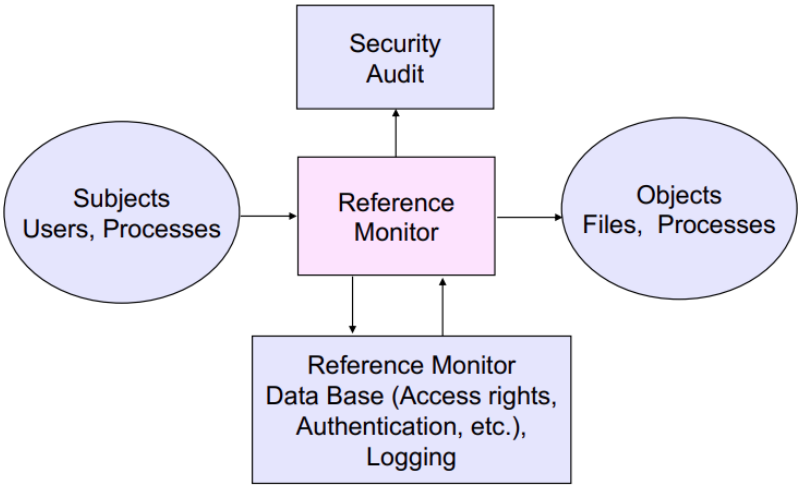

Abbildung 89: Security Architecture

Sämtliche Zugriffe im System führen über den Reference Monitor der auch weiß wer was auf welchem Objekt tun darf. Weiters produziert der Reference Monitor ein Security Audit um festzuhalten was mit welchem Objekt passiert. Der Reference Monitor sitzt sozusagen zwischen Subjekten und Objekten.

10.13 Zusammenfassung

- Security ist nicht nur eine technische Frage
  - beinhaltet policies in Firmen
  - Wie gehen wir mit Information um?
- Ziele:
  - Confidentiality
  - Integrity

  - Availability
- Bedrohungen
- Design Prinzipien
  - Verschlüsselung ist nur bei offenen Systemen notwendig
- Schutzmechanismen im Betriebssystem
  - Access Matrix als Basis, hat allerdings verschiedene Ausprägungen oder Implementationen in Betriebssystemen. Regelt Zugriffe zwischen Subjekten und Objekten.