

VO Operating Systems V1

16.06.2025

Matti Fischbach
matti.fischbach@web.de

*Erstellt mit den hervorragenden Foliensätzen von Peter Thoman
und den Foliensätzen von Herr Radush,
die die Lehrveranstaltung Betriebssysteme
an der Universität Innsbruck leiten.*

Jegliche Informationen stammen aus den Foliensätzen der Lehrveranstaltung
und demnach der zugrundeliegenden Quelle:

*Operating System Concepts
von Abraham Silberschatz, Peter Bare Galvin und Greg Gagne
(Tenth Edition)*

Contents

1	Betriebssystem	5
1.1	Definition:	5
1.2	Ziele:	5
2	Struktur eines Computersystems	5
2.1	Hardware	5
2.2	Anwendungsprogramme	5
2.3	Benutzer	5
2.4	Userinterface	5
3	Aufbau	5
3.1	Kernel	5
3.2	System services	6
3.3	Middleware	6
3.4	Organistation	6
3.5	Betriebssystem Ablauf	6
4	I/O Operation	7
5	Interrupts und Interrupt based Systems	7
5.1	direct Memory-Access	8
5.2	Dualmode	8
5.3	Timer	9
6	Computer Organisation	9
6.1	Speicherhierarchie	9
6.2	Von Neumann	10
6.3	Gemeinsame Speichersysteme	10
6.4	Clustersysteme	10
6.5	Multiprozess Programmierung und Multitasking	10
7	Ressourcen Verwaltung	11
7.1	Prozessverwaltung	11
7.2	Speicherverwaltung (flüchtig)	11
7.3	Dateisystemverwaltung (nicht flüchtig)	11
7.4	Massenspeicherverwaltung	11
7.5	Cacheverwaltung	11
7.6	Zusammenfassung	12
7.7	Schutz und Sicherheit	12
7.8	Virtuallisierung	12
7.9	Verteilte Systeme	12
8	Systemcalls	12
8.1	POSIX	13
8.2	Systemcall Arten	13
8.3	Microkernel	14
9	Prozesskonzept	14
9.1	Prozess	14
9.2	Speicherstruktur	14
9.3	Zustand	14
9.4	Prozesskontrollblock	15
9.5	Linux	15
10	Prozessplanung	15

10.1	Queues	16
10.2	Contextswitch	16
11	Prozessoperationen	16
11.1	Prozesserstellung	16
11.2	Prozessterminierung	17
12	Threads	18
12.1	Single vs. Multithreaded Programme	18
12.2	Anwendung (Server)	18
12.3	Concurrency vs. Parallelism	18
12.4	Parallelism	19
12.5	Data vs Task Parallelism	19
12.6	User- und Kernelthreads	19
12.7	Multithreading Modelle	20
12.8	PThread (POSIX Threads)	20
12.9	fork und exec	21
12.10	Signals und Interrupts	21
12.11	Abbruch	21
13	Interprozess Kommunikation (IPC)	21
13.1	Fundamentales Model	22
13.2	Messagepassing	22
13.3	Unnamed Pipe	23
13.4	Named Pipe (FIFO)	24
13.5	Messagequeue	24
13.6	Shared Memory	26
14	Synchronisation	27
14.1	Data Hazards	28
14.2	Critical Sections	29
14.3	Atomics	29
14.4	Atomic Operationen	29
14.5	Shared Memory mit Synchronisation	29
14.6	Mutex	30
14.7	Dekker's Algorithmus	31
14.8	Peterson's Algorithmus	31
14.9	Memorybarrier	32
14.10	Deadlocks	32
14.11	Bankieralgorithmus	33
14.12	Deadlock Detection	34
14.13	Deadlock Behebung	35
14.14	Livelocks	35
14.15	Condition Variables	36
14.16	Semaphores	37
14.17	Ring Buffer	38
14.18	Producer - Consumer Problem	39
14.19	Dining Philosophers Problem	40
14.20	Barriers	41
14.21	Interrupt-based Synchronisation	42
15	Synchronisation (Hardware)	42
15.1	Spinlock / Busy waiting	43

15.2	Semaphoren	43
15.2.1	Implementierung mittels Spinlock:	43
15.2.2	Implementierung ohne Spinlock:	44
15.3	Monitor und Condition Konstrukt (Java)	45
16	Alternative Ansätze der Synchronisation	46
16.1	Transactional Memory	47
17	Input/Output (I/O)	47
17.1	Architektur	47
17.2	Speicher	48
17.3	I/O Bus	48
17.4	Device Communication	48
17.4.1	Beispiel <i>Memory Mapped I/O</i> an einem Arduino Uno ATmega328P:	49
17.4.2	Beispiel <i>DMA-Buffer</i> :	49
18	Speicher Hardware und Software	50
18.1	HDD	50
18.2	SSD	51
18.3	NAND und NOR Flash	51
18.4	RAID	51
18.5	RAID Standard Levels	52
18.5.1	RAID 0	52
18.5.2	RAID 1	52
18.5.3	RAID 2	53
18.5.4	RAID 3	53
18.5.5	RAID 4	53
18.5.6	RAID 5	54
18.5.7	RAID 6	54
18.6	RAID Hybrid Levels	54
18.6.1	RAID 01 / RAID 10	54
18.6.2	RAID 50	55
19	Filesystems und Partitioning	55
19.1	Partitiontable	55
19.1.1	MBR	55
19.1.2	GPT	55
19.2	FAT Filesystem	56
19.3	Journaling Filesystem	56
19.4	Multi-Disk Filesystem	56
20	Scheduling	56

1 Betriebssystem

1.1 Definition:

- Software zur Verwaltung der Computerhardware
- Low-Level Grundlage der Anwendungsprogramme
- Interface zwischen User und Hardware

1.2 Ziele:

- Ausführen von User-Programmen
- Benutzerfreundliche/einfache Lösung von Problemen (*higher level programming*)
- effiziente Nutzung der Hardware

2 Struktur eines Computersystems

2.1 Hardware

grundlegende Ressourcen wie:

- CPU,
- GPU,
- RAM,
- usw.

2.2 Anwendungsprogramme

- Verwenden Ressourcen der Hardware um Probleme zu Lösen
- Programme wie Textverarbeitung, usw.

2.3 Benutzer

- Mensch, Maschinen bzw. andere Computer

Das Betriebssystem stellt wie bereits angesprochen die Schnittstelle zwischen Benutzer und Hardware bereit. Dabei stellt es eine angemessene Performance, security und Benutzerfreundlichkeit bereit. Es kann in Ein- bzw. Mehrfachbenutzergeräten und in Embeddedsystems verwendet werden. Hier stellt es die Hardware Ressources und die Kontrolle für die einzelnen Prozesse bereit.

2.4 Userinterface

- CLI (Command Line Interface)
- GUI (Graphical User Interface)

3 Aufbau

3.1 Kernel

- Startet mit als erstes beim Einschalten des Computers
- läuft im Hintergrund bis zum Abschalten des Systems

Es wird zwischen Präemptiven und Kooperativen (nicht präemptiven) Kernels unterschieden:

- **Präemptiv:**

Ein laufender Prozess kann von dem Betriebssystem unterbrochen werden, auch wenn er nicht *freiwillig* den Prozessor freigibt. Dies ist notwendig, wenn eine Prozess mit höherer Priorität bereit ist auf dem Prozessor zu laufen.

- **Kooperativ:**

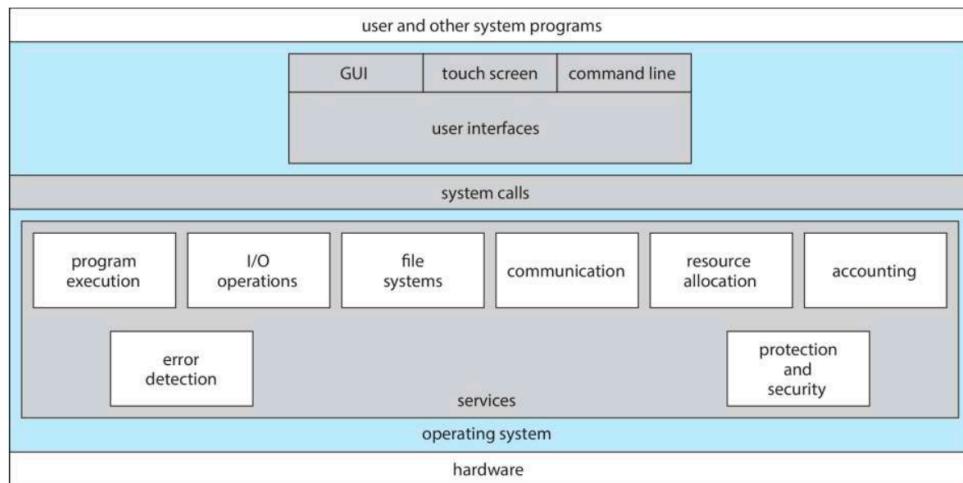
Ein laufender Prozess kann **nicht** von dem Betriebssystem unterbrochen werden. Dieser muss *freiwillig* den Prozessor verlassen. Dies bringt eine langsamere Reaktionszeit mit sich, zugrunde einer geringeren Komplexität. Bei schlechter Programmierung kann ein Prozessor langfristig *verstopfen*.

3.2 System services

- Systemprogramme und Deamons (*im Hintergrund laufende Prozesse, welche meist für den Benutzer nicht direkt sichtbar sind*)
- Treiber
- Systembibliotheken (*Libraries*)
- GUI
- CLI Schnittstelle

3.3 Middleware

- erleichtern die Anwendung (.NET, ...)
- Datenbanken
- Grafik (X-Window System, ...)



3.4 Organistation

Die CPU und der Gerätekontroller ist durch den Systembus mit dem Speicher (RAM) verbunden. Die Treiber der einzelnen Geräte ist die Schnittstelle zwischen der wirklichen Hardware bzw. dem Gerätekontroller und dem restlichen System über den Systembus. Dadurch wird ein gemeinsamer Zugriff auf den Speicher ermöglicht. Durch Interrupts wird eine gleichzeitige Ausführung von der CPU und den Geräten ermöglicht.

3.5 Betriebssystem Ablauf

Bootstrapping:

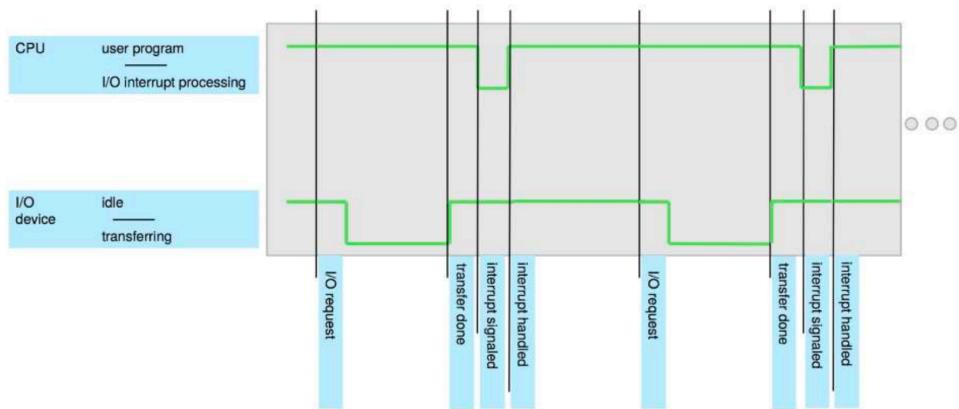
1. Bios: (ROM)
2. Systeminitialisierung

3. Kernel wird geladen
4. Start der Systemdienste

4 I/O Operation

Das Programm stellt eine I/O Request über das Betriebssystem an den Gerätetreiber und lädt die benötigten Register. Der Gerätekontroller prüft die Register und führt die angefragte Aktion aus (*z.b. Zeichen von Tastatur lesen*). Die Daten werden in einem lokalen Buffer gespeichert und der Kontroller informiert über den Gerätetreiber das die Aktion abgeschlossen wurde. Nun findet ein Interrupt ausgelöst vom Systembus statt, was die Kontrolle an das System zurückgibt. Die Daten (und Statusinformationen) können gelesen werden.

Timeline:



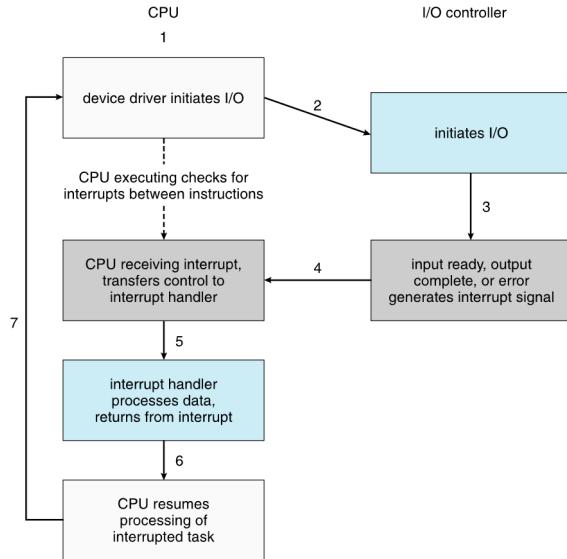
5 Interrupts und Interrupt based Systems

Werden vom Gerätekontroller ausgelöst. Nach jedem CPU Cycle wird überprüft ob ein Interrupt vorliegt. Der Interrupt wird von der CPU abgefangen und an einen Interrupthandler weitergeleitet. Der Interrupthandler speichert die Adresse des Unterbrochenen Befehls und die Register der CPU. Weiter wird der Interrupt verarbeitet. Schlussendlich wird der Zustand vor dem Interrupt wieder hergestellt. Jeder Interrupt hat einen Interruptvektor. Dieser beinhaltet die Interrupt-Zahl, die Adresse des Interrupthandlers und die Interruptchain. Es gibt *Unmaskierte* (*nicht ausschaltbar, z.b. Speicherfehler*) und *Maskierte* (*ausschaltbar*) Interrupts. Interrupts haben unterschiedliche Prioritäten.

Interrupttypen:

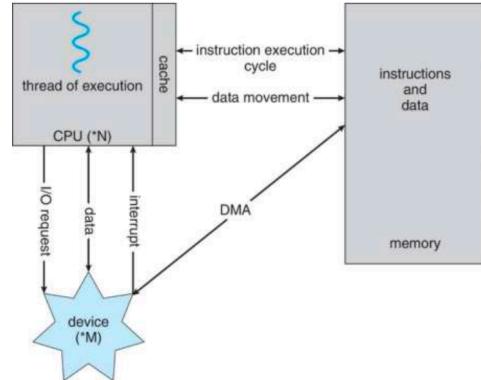
- Hardwareinterrupts (Geräte)
- Softwareinterrupts (Exception, Trap, Error, Syscall, Segfault, ...)

Interrupt-cycle (I/O):



5.1 direct Memory-Access

Zwischen nichtflüchtigem Speicher und dem Gerätekontroller. Wird genutzt um Daten in Blöcken direkt vom Buffer des Kontrollers im Hauptspeicher abzulegen. Der Abschluss wird der CPU durch einen Interrupt kommuniziert. Es findet eine Unterbrechung pro **Block** und **NICHT** pro **Byte** statt. Die CPU ist nicht beteiligt.



5.2 Dualmode

Bietet verbesserte Security. Es wird ein **Modus-Bit** benutzt um zu signalisieren ob der:

- Kernelmode (0),

oder der

- Usermode (1)

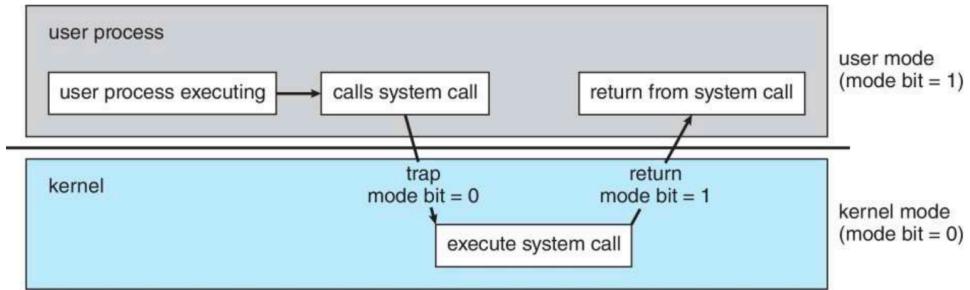
verwendet wird.

Kernelmode: (auch Privilegedmode, Systemmode, Supervisormode)

Startet den Kernel, ist verantwortlich für Systemaufrufe und Interrupts. Lässt Priviligierte Anweisungen zu.

Usermode:

- Usercommands-/code



5.3 Timer

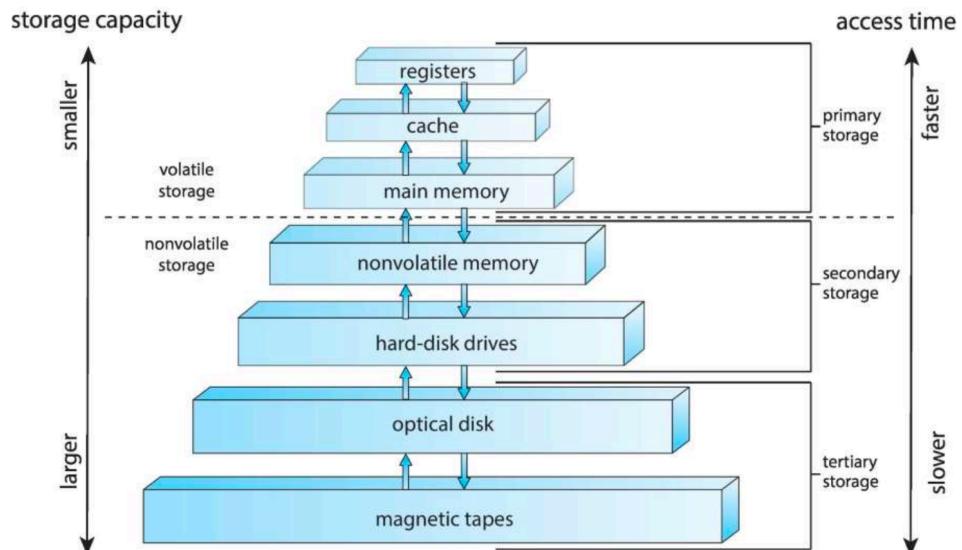
Timer werden genutzt um die Beanspruchung von Ressourcen für bestimmte Prozesse zu begrenzen. Nach einer bestimmten Zeit wird ein Interrupt getriggert um die Kontrolle wieder zu erlangen. Dies kann verhindern, dass zum Beispiel Endlosschleifen die CPU Ressourcen blockieren. Dies kann durch privilegierte Anweisungen unterbunden werden.

Fixed Timer: fixe Zeitspanne (z.B. $\frac{1}{60}$ sec)

Variable Timer: Clockfrequency oder Counter

6 Computer Organisation

6.1 Speicherhierarchie

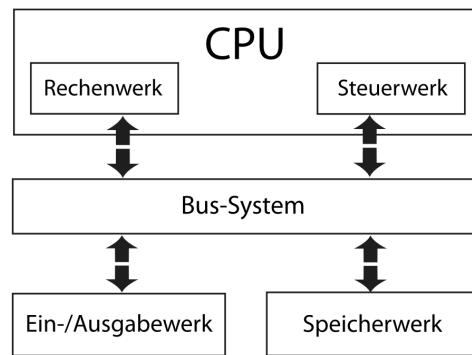


6.2 Von Neumann

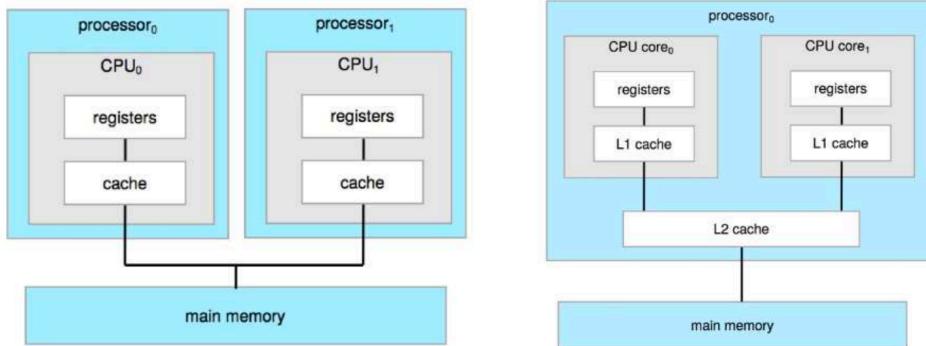
- Bus-System
- Memory

Der Hauptspeicher (RAM) ist das einzige von der CPU direkt zugreifbare Speichermedium. Wird meist als DRAM implementiert und ist flüchtig und wieder beschreibbar.

Der Sekundärspeicher wird in einzelne logische Sektoren unterteilt die vom Speicherkontroller dann die Interaktion mit dem Computer ermöglicht.



6.3 Gemeinsame Speichersysteme



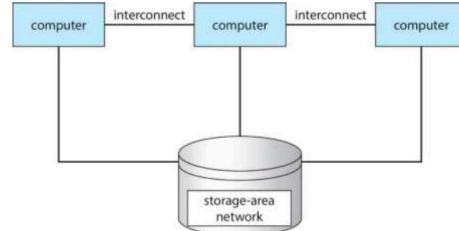
6.4 Clustersysteme

High availability:

- Error tolerant
- (A-)Synchronous Clustering
- Load splitting

High performance:

- Parallel
- speed



6.5 Multiprozess Programmierung und Multitasking

Mehrere Benutzer können System parallel benutzen. Dabei können Programme(/Prozesse) gleichzeitig rechnen. Dies ist besonders nützlich bei Planung, Warteschlangen, Prioritäten, und generellem Lastausgleich, da die Ressourcennutzung maximiert wird. Auch bei jeglicher I/O ist eine Multiprocessing Ansatz sinnvoll, da Anfragen *gleichzeitig* bearbeitet werden.

Multiprocessing und *Multitasking* sind unterschiedlich, da hier auf einem 1 Benutzersystem mehrere Prozesse/Anwendung gleichzeitig laufen.

7 Ressourcen Verwaltung

7.1 Prozessverwaltung

Prozess: Ist ein Programm in Ausführung und eine Arbeitseinheit in dem System. Die Ressourcen (CPU, Memory, I/O) stehen dem Prozess zur Verfügung und werden nach der Beendigung freigegeben.

Betriebssystem: Ist verantwortlich für das erstellen, löschen und unterbrechen der Prozesse, sowie der Zuordnung der Ressourcen. Es ist außerdem verantwortlich für die Synchronisation und der Inter-Prozesskommunikation. Außerdem ist es verantwortlich für die Deadlock Behbung/Behandlung.

Singlethread Prozess: Anweisungen werden sequentiell ausgeführt und in einem Programmcounter wird der jeweilig nächste Schritt/Anweisung gespeichert.

Multithread Prozess: Hier hält jeder Prozess seinen eigenen Programmcounter. Die Programmierung muss parallelisiert werden

Andere: Userprozesse, OS-Prozesse, usw.

7.2 Speicherverwaltung (flüchtig)

Inhalt: Die Gesamtheit bzw. Teile des Programmcodes/Maschinencodes. Nötig um das Programm schlussendlich auszuführen. Außerdem alle weiteren Daten die das Programm benötigt, wie z.b. heap, stack, filehandles usw.

Das Betriebssystem verfolgt die Speicherverwendung und ist zuständig für Speicher Zuordnung und Freigabe. Außerdem ist es zuständig für die Koordination vom Laden und Freigeben von Daten aus dem Sekundärspeicher.

7.3 Dateisystemverwaltung (nicht flüchtig)

Daten werden häufig in Dateisystemen gespeichert. Sie bieten eine einheitliche und logische Ansicht der Informationsspeicherung. Eine Datei ist hier eine logische abstrakte Einheit. Die Daten des Dateisystems werden in der Regel auf dem Sekundär- bzw. Tertiärspeicher gespeichert.

Das Betriebssystem ist zuständig für die Verwaltung und Organisation, der Zugriffskontrolle, dem Erstellen und Löschen, dem Bearbeiten und dem Zuordnen im nichtflüchtigen Speicher.

7.4 Massenspeicherverwaltung

Weiterhin werden Daten oft temporär(USB-Stick) oder langfristig(Backup-Drive) gespeichert. Dies ist entscheidend für die Geschwindigkeit des Computerbetriebs, da mehr Daten = mehr Arbeit für das System.

Das Betriebssystem ist zuständig für das Mounten, Unmounten, Freespace managing, Zuteilung, I/O Planung, Partitionierung und dem Schutz der Datenträger bzw. Daten.

7.5 Cacheverwaltung

Der Cachespeicher ist ein vorübergehender Speicher für das Speichern bei Datenkopien. Der Cache ist meist wesentlich kleiner als der Speicher(RAM) und besteht aus Registern, Befehlscache und Datencache.

Die Hardware ist für die Verwaltung der Cachezeilen, Cachetreffer(Lokalität), Cachefehler, Cachekohärenz(Multiprocessing) und der Cacheersatzrichtlinien zuständig.

7.6 Zusammenfassung

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Die Übertragung ist in der Reihenfolge:

secondary mem → main memroy(RAM) → cache → hardware registers

7.7 Schutz und Sicherheit

Das Betriebssystem schützt die Daten mit Zugriffskontrolle (*only specific Users and Groups*), Verteidigung gegen Angriffe bzw. unerwünschten Zugriff.

7.8 Virtualisierung

“Betriebssystem in Betriebssystem”

Oft auch als Emulation bekannt, wenn ein anderes (Gast-)Betriebssystem in einem (Host-)Betriebssystem genutzt wird. Nützlich um Programme auf anderen Plattformen zu testen bzw. zu entwickeln.

7.9 Verteilte Systeme

Systeme die durch eine Schnittstelle miteinander verbunden sind. Meistens über TCP/IP bzw. über LAN, WAN, MAN oder PAN. Die Kommunikation findet durch Datenaustausch statt. Hier gibt es sogenannte Netzwerkbetriebssysteme die die Illusion eines einzigen Systems mit vielen einzelnen Computern simuliert.

Beispiele: Traditionell (Server, Client); Peer-To-Peer (Clients that form a system); Cloud Computing; usw.

8 Systemcalls

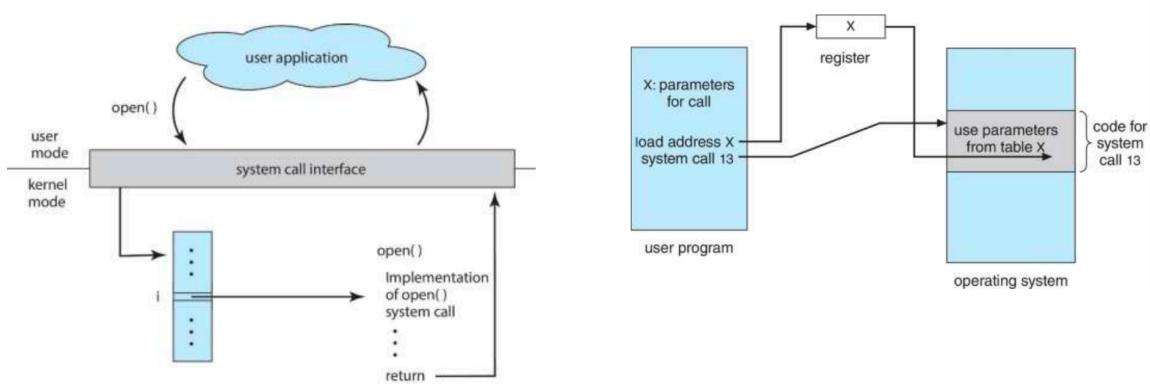
Systemcalls bieten eine Schnittstelle zwischen dem Programmierer und dem Betriebssystem. Sie ermöglichen Low-Level Befehle einfach in Higher-Level Programmiersprachen zu verwenden. Beispiel ist die Win32-API auf Windows oder die POSIX-API für UNIX. Aber auch die Java-API für die Java Virtuelle Maschine.

Für die Verwendung von Systemcalls werden Register und der Stack genutzt um Daten bzw. Parameter an den jeweiligen Aufruf zu übergeben.

Unter einem Modularen Betriebssystem werden die Systemcalls zusätzlich zum Betriebssystem gespeichert. Somit ist die Schnittstelle Modular und kann nach belieben angepasst werden.

8.1 POSIX

POSIX ist eine Standardisierte Programmierschnittstelle zum Betriebssystem UNIX. Sie folgt dem ISO/IEC/IEEE 9945 Standard. Die `libc` Bibliothek folgt dem POSIX Standard. UNIX alleine bietet Basis Definitionen, Konventionen und Konzepte. Sie ist die schlussendliche System-Schnittstelle mit C-Systemaufrufen und Header-Dateien.



8.2 Systemcall Arten

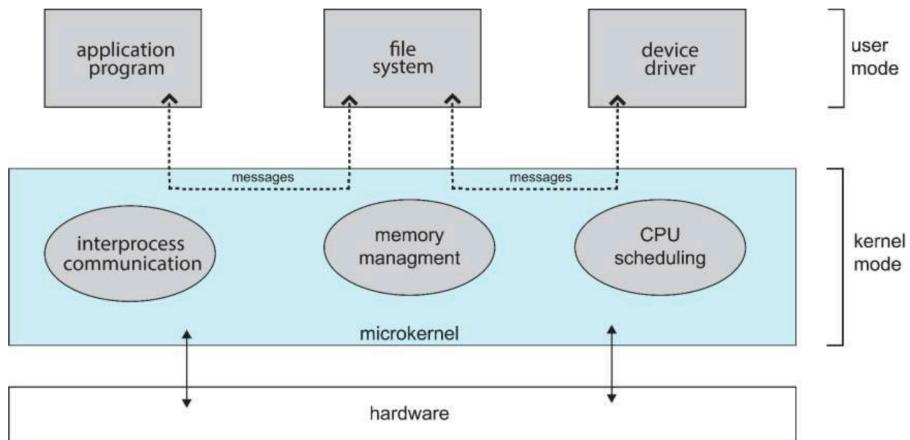
Prozesssteuerung: Zum Prozess Erstellen, Löschen, Laden und Ausführen, Abrufen und Festlegen von Attributen, Warte- bzw. Signalereignisse, Speicher Zuweisung und Freigabe, core dump, Debugger und Locking für gemeinsame Daten.

Dateiverwaltung: Zum Erstellen, Löschen, Öffnen und Schließen von Dateien, Lesen, Schreiben und Bearbeiten von Daten in Dateien und das Abrufen von Dateiattributien.

Devicemanagement: Zum Anfordern und Freigeben von Geräten, Lesen, Schreiben und Bearbeiten von Geräten, Abrufen und Festlegen von Gerät Attributien, sowie das logische An- und Abkoppeln von Geräten

Diese Unterschiedlichen Arten von Systemcalls werden verwendet um bereits vorhandene Informationen zu nutzen bzw. festzulegen. Sie sind wichtig um den Zugriff zu schützen und die Kommunikation zwischen Prozessen zu organisieren.

8.3 Mikrokernell

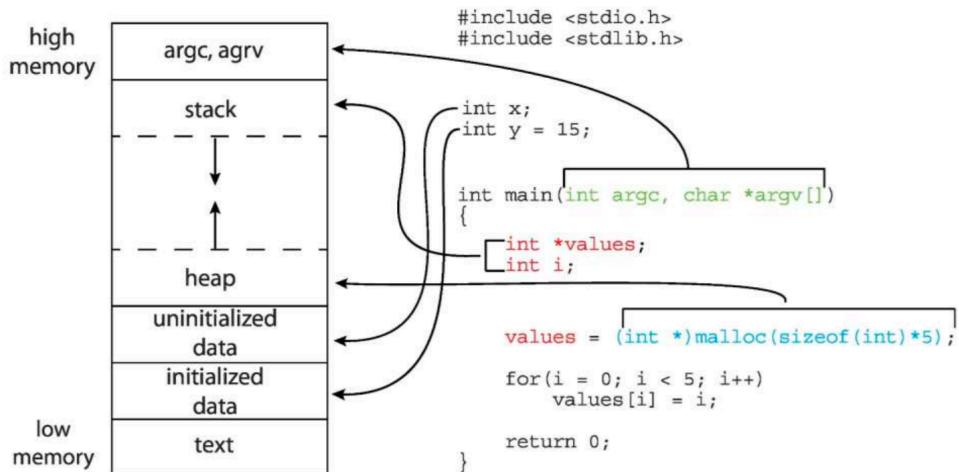


9 Prozesskonzept

9.1 Prozess

Ein Prozess ist eine Ausführbare Datei auf der Festplatte die Programmcode und/oder Textabschnitte enthält. Die Datei wird geladen und sequenziell ausgeführt. Hier hat der Prozess wie bereits weiter oben angesprochen einen Programcounter, Prozessorregister, einen Stack, globale Daten wie Variablen und einen Heap auf dem Speicher dynamisch alloziert werden kann.

9.2 Speicherstruktur



9.3 Zustand

- **new:** wird gerade erstellt
- **ready:** bereit für Zuweisung (auf Prozessor)
- **running:** Ausführen der Anweisungen
- **waiting:** warten auf Ereignis
- **finished:** Ausführung abgeschlossen

9.4 Prozesskontrollblock

- Zustand
- Programcounter
- CPU-Register
(Data in process dependant registers)
- CPU-Schedulinginformation
(Priority, Schedulingqueue pointer)
- Speicherverwaltungsinformationen
(Registers, Pagetable)
- Abbrechungsinformation
(CPU usage, Time, Timelimits)
- I/O Statusinformation
(Assigned Devices, open files)

process state
process number
program counter
registers
memory limits
list of open files
...

Jeder Thread hat auch einen eigenen Kontrollblock!

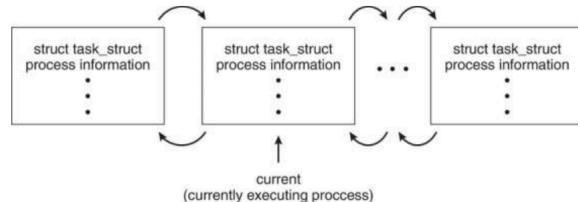
9.5 Linux

- **task_struct** Datenstruktur unter `<include/linux/sched.h>`

```
pid t_pid;          /* Prozessbezeichnung */
long state;         /* Zustand */
unsigned int time_slice; /* Zustandsinformation */
struct task_struct *parent; /* Vaterprozess */
struct list_head children; /* Kinderprozessen */
struct files_struct *files; /* Offene Dateien */
struct mm_struct *mm;    /* Speicher */
```

- **Prozesstabelle**

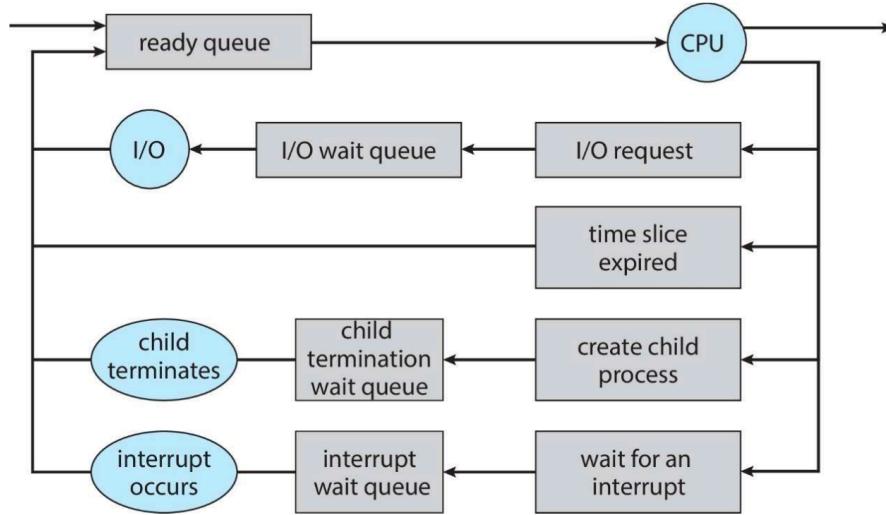
- `current->state = ...`



10 Prozessplanung

Die Prozesse in der Prozessqueue werden von Prozessplaner einem CPU-Kern zugeordnet. Dies ist sinnvoll um die CPU-Auslastung zu maximieren und Prozesse möglichst effizient und schnell einem CPU-Kern zuzuordnen. Dabei muss die Prozessqueue verwaltet werden und die einzelnen CPU-Kerne, sowie I/O und Interrupts auf Bereitschaft überprüft werden. Oft gibt es verschiedenen Warteschlangen zwischen denen gewechselt werden muss.

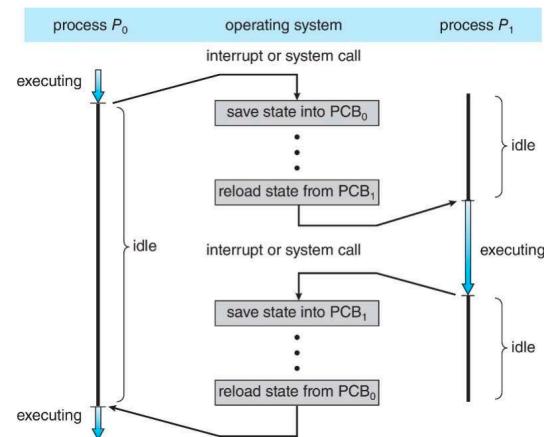
10.1 Queues



Prozesse die bereit sind auf der CPU ausgeführt zu werden, warten in der Ready-Queue darauf auf der CPU ausgeführt zu werden. Ist "Platz" auf der CPU wird ein neuer Prozess aus der Ready-Queue auf der CPU ausgeführt. Während dem Ausführen des Prozesses kann es dazu kommen das dieser die CPU wieder verlässt. Dies passiert hier wenn eine I/O Anfrage anfällt, der Timer des Prozesses abgelaufen ist, ein Kindprozess erstellt werden soll oder ein Interrupt die Ausführung unterbricht. Wenn diese Anfragen/usw. verarbeitet wurden wird der Prozess erneut in die Ready-Queue eingefügt. Dies passiert solange bis der Prozess fertig ausgeführt ist und anstatt erneut in die Ready-Queue zu gelangen den Kreislauf verlässt.

10.2 Contextswitch

Bei dem Kontext handelt es sich um die CPU Register, Zustand und Speicherinformation welche im Prozesskontrollblock gespeichert sind. Bei einem Kontextswitch wird der Kontrollblock eines Prozesses gespeichert und der eines anderen/neuen geladen. Wie bereits oben erklärt finden *ständig* Contextswitches statt, hervorgerufen durch die oben erklärten Anfragen/Interrupts/usw. .

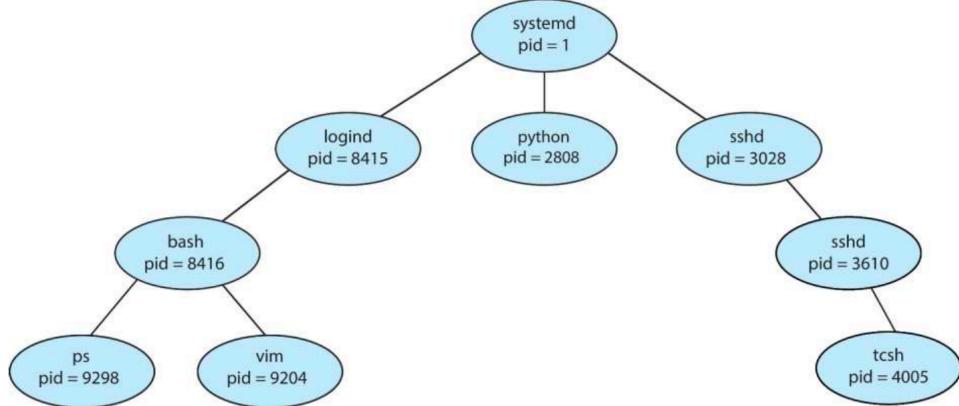


11 Prozessoperationen

11.1 Prozesserstellung

Bei der Prozesserstellung erstellt ein (jetzt Eltern-)Prozess einen Kindprozess. Dieser hat eine Prozessid (PID bzw. pid_t (POSIX)). Es wird festgelegt ob die Daten des Elternprozesses (teilweise) geteilt werden oder kopiert werden. Außerdem kann festgelegt werden ob auf den

Kindprozess schlussendlich gewartet werden soll oder ob dieser von alleine schließt, bzw. ein Daemon ist. Somit wird ein Prozessbaum erstellt. Beispiel in Linux:



Die Erstellung eines Kindprozesses unter UNIX folgt folgendem Schema:

```

#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
    pid_t pid = fork();           /* Kindprozess erstellen */
    if (pid < 0) {
        fprintf(stderr, "Fork failed.");
        return 1;
    } else if (pid == 0)          /* Kindprozess */
        execlp("/bin/ls", "ls", NULL);
    } else {                      /* Elternprozess */
        wait(NULL);              /* Kindsbeendigung warten */
        printf("Child completed.");
    }
    return 0;
}
  
```

`fork()` erstellt einen Kindprocess als Duplikat des Elternprozesses.

`exec()` lädt neues Programm in dem Kindprozess. Dieses ersetzt den Speicherplatz des Kindprozesses.

`wait()` wartet auf die Beendigung des Kindprozesses

11.2 Prozessterminierung

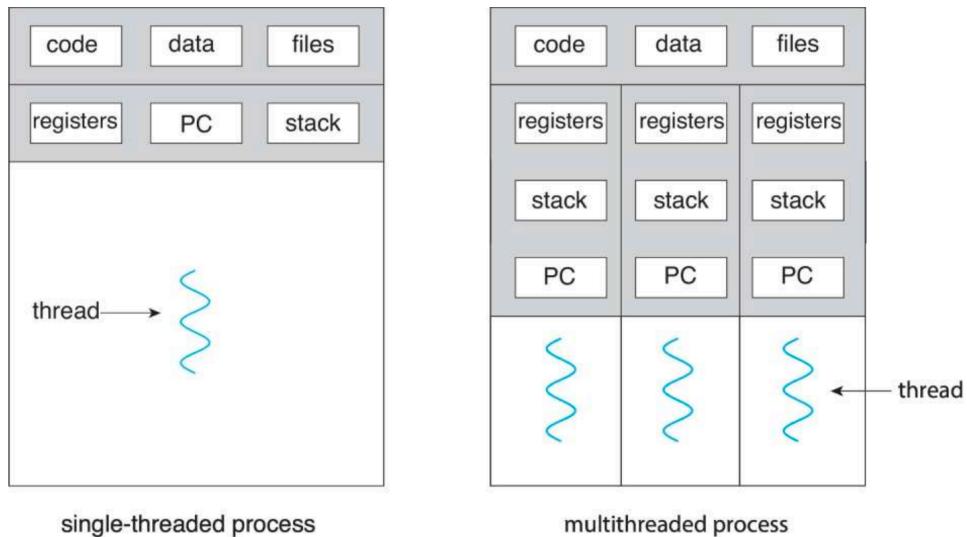
Ein Prozess terminiert wenn entweder die letzte Anweisung ausgeführt wird oder `exit(int status)` im Kindprozess aufgerufen wird. `wait(int *status)` wartet auf die Beendigung des Kindes und kann den exit status code des Kindes abrufen. `abort()` terminiert den Kindprozess vom Elternprozess aus. Wird nicht auf die erstellten Kindprozesse gewartet werden entweder alle nicht terminierten Kinder vom Betriebssystem terminiert (Cascading) oder es verbleiben verwaiste Kindprozesse. Terminiert ein Kind ohne das der Elternprozess auf diesen wartet nennt man das Zombieprozess.

12 Threads

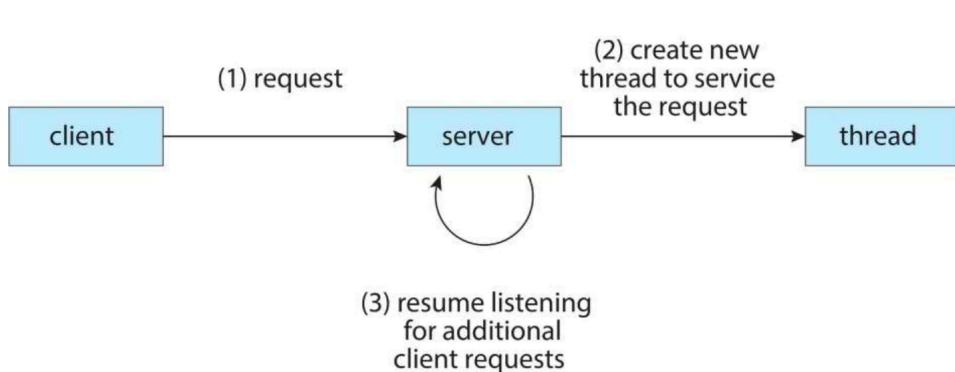
Da viele moderne und Interaktive Anwendungen mehrere Aufgaben parallel ausführen wurde nach einer neuen Methode gesucht um Daten parallel zu verarbeiten. Dafür bieten sich Threads an. Diese sind lightweight und kompatibel mit neuen Multithread-Kernels und Multithread Prozessoren.

Die Vorteile von Threads sind verbesserte Reaktionsfähigkeit auch wenn Teile des Prozesses Blockieren, was besonders wichtig für Grafische Benutzeroberflächen ist. Außerdem lassen sich die Ressourcen eines CPU-Kerns und generell die Ressourcen besser teilen, da kein automatisches Kopieren der Daten des Prozesses stattfindet. Die Threadingerstellung ist zudem wesentlich günstiger und Contextswitches bereiten weniger Overhead, da immer noch auf dem selben Prozessorkern gerechnet wird. Multithread Anwendungen sind zudem besser skalierbar auf neuen Multicore Systemen.

12.1 Single vs. Multithreaded Programme



12.2 Anwendung (Server)



12.3 Concurrency vs. Parallelism

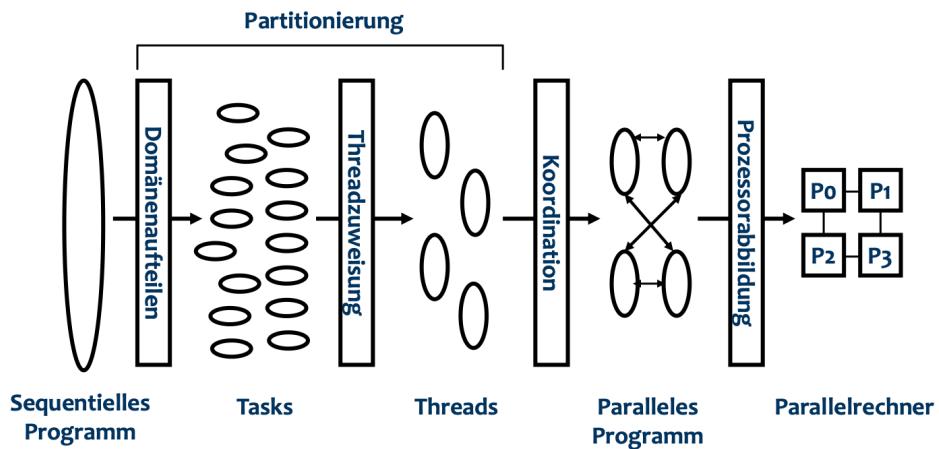
Concurrency:

$$T_1 = \text{Task 1}; T_2 = \text{Task 2}; \\ T_1 \rightarrow T_2 \rightarrow T_1 \rightarrow T_2 \rightarrow T_1 \rightarrow T_2 \rightarrow \dots$$

Parrallelism:

$$T_1 = \text{Task 1}; T_2 = \text{Task 2}; \\ T_1 \rightarrow T_1 \rightarrow T_1 \rightarrow \dots \\ T_2 \rightarrow T_2 \rightarrow T_2 \rightarrow \dots$$

12.4 Parallelism



12.5 Data vs Task Parallelism

Bei der Datenparallelisierung findet die selbe Berechnung auf unterschiedlichen Daten statt.
Bei der Taskparallelisierung finden unterschiedliche Berechnungen auf den gleichen bzw. verschiedenen Daten statt.

Bei der Datenparallelisierung wird ein Datensatz in kleinere Teilstücke zerlegt. Diese können mit dem gleichen Code von verschiedenen Prozessen und/oder Threads verarbeitet werden. Dies findet häufig bei Matrix Berechnungen oder Machine Learning statt.

Bei der Taskparallelisierung werden unterschiedliche Aufgabe/Berechnungen auf demselben oder unterschiedlichen Daten ausgeführt. Dies ist häufig bei komplexen Problemen mit mehreren Arbeitsschritten und ideal wenn das Gesamtproblem in einzelne kleinere Teilprobleme zerlegt werden kann. Betriebssysteme verwenden häufig diesen Ansatz um zum Beispiel gleichzeitig Daten zu verarbeiten und auf der Grafischen Oberfläche anzuzeigen.

12.6 User- und Kernelthreads

Userthreads:

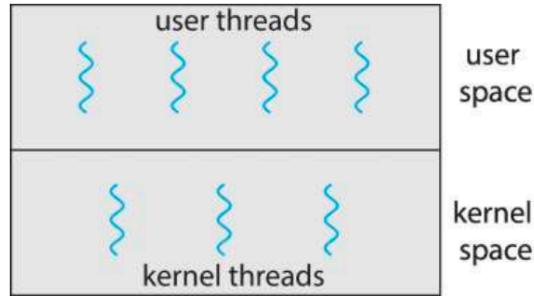
Verwaltung durch Schnittstelle des Betriebssystems:

- POSIX: pthread
- Windows: Threads
- Java: Threads

Kernelthreads:

Verwaltung durch den Kernel

- Linux
- Windows
- MacOS
- ...



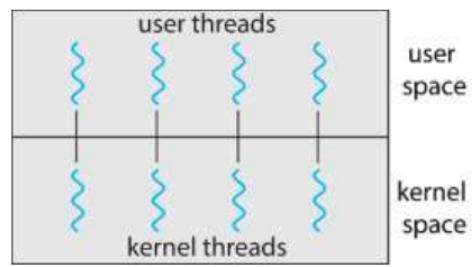
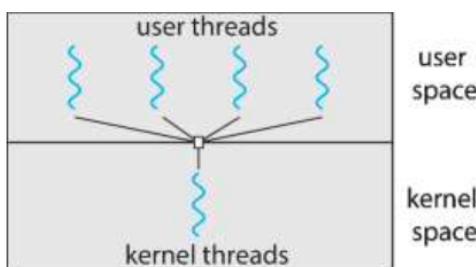
12.7 Multithreading Modelle

Many-to-One:

Meherere Benutzer werden einem einzelnen Kernel-Thread zugeordnet. Das Blockieren eines Threads blockiert alle anderen. Dadurch ist möglicherweise keine Parallelisierung möglich. Dieser Ansatz wird von wenigen Betriebssystemen genutzt

One-to-one:

Jedem Benutzer wird ein Kernel-Thread zugeordnet. Das bietet mehr Parallelität bringt jedoch den Nachteil das die Anzahl der Threads pro Benutzer möglicher begrenzt ist. Linux und Windows benutzen diesen Ansatz.

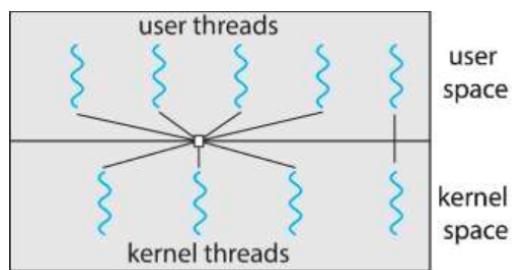
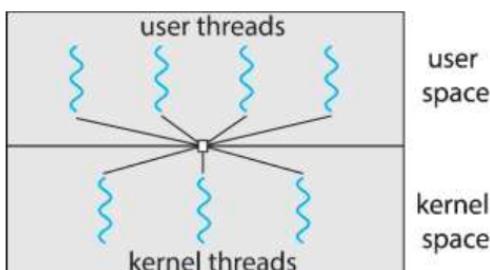


Many-to-Many:

Viele Benutzer-Threads werden vielen Kernel-Threads zugeordnet. Dies benötigt eine ausreichende Anzahl an Kernel-Threads. Dieser Ansatz wird von Windows mit Thread-Fiber-Package verwendet, wird jedoch nicht häufig verwendet.

2-Layers:

Ähnlich wie der Many-to-Many Ansatz. Bindung von Benutzer-Threads an Kernel-Threads möglich, wenn in Benutzung.



12.8 PThread (POSIX Threads)

Wie bereits weiter oben angesprochen ist pthread die POSIX Schnittstelle für Threads auf UNIX Systemen.

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* geteilte Variable */
int sum;

/* Threadfunktion */
void *runner(void *param) {
    int i, n = atoi(param);
    sum = 0;
    for(i = 1; i <= n; i++)
        sum += i;
    pthread_exit(0);
}

int main(int argc, char *argv[]) {
    /* Thread-Bezeichner */
    pthread_t tid;

    /* Thread-Attribute einstellen */
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    /* Thread erstellen */
    pthread_create(&tid, &attr,
                  runner, argv[1]);

    /* Thread-Beendigung warten */
    pthread_join(tid, NULL);
    printf("sum = %d\n", sum);
}

```

12.9 fork und exec

fork dupliziert den aufrufenden Thread bzw. den Prozess der den Thread gestartet hat. exec ersetzt den laufenden Prozess und somit auch alle erstellten Threads.

12.10 Signals und Interrupts

Signale, die erzeugt durch ein bestimmtes Ereignis und an einen Prozess übermittelt, werden vom Kernel Signalhandler bzw. einem Benutzerdefinierten Signalhandler verarbeitet. Die Übermittlung an die Threads des Prozesses findet entweder für alle Threads, bestimmte Threads oder einen bestimmten Thread für alle Signale statt.

12.11 Abbruch

Es wird zwischen Asynchronen und Verzögerten Abbrüchen unterschieden. Ein Asynchroner Abbruch beendet den Thread sofort, wohingegen ein verzögerter Abbruch bei bestimmten Abbruchpunkten beendet.

Für den Abbruch benutzen wir `pthread_setcancelstate(...)` und für den Typen `pthread_setcanceltype(...)`. Mit `pthread_testcancel()` wird überprüft ob der Thread abgebrochen werden soll. `pthread_cancel(thread_id)` bricht den Thread mit der entsprechenden Id ab. Auch Threads die abgebrochen werden müssen mit `pthread_join(tid, NULL)` wieder mit dem main thread bzw. dem Elternprozess verbunden werden. Die Abbruch Typen sind Off/NULL, Deferred und Asynchronous.

13 Interprozess Kommunikation (IPC)

Die Interprozess Kommunikation wird benötigt, wenn ein Problem in mehrere Teilprobleme zerlegt wird (*performance*) und von mehreren Prozessen gleichzeitig bearbeitet wird. Hier ist es nicht immer möglich die einzelnen Aufgaben vollständig auf die einzelnen Prozesse aufzuteilen und es ist nötig Daten zu bestimmten Zeitpunkten zwischen den arbeitenden Prozessen zu teilen um schlussendlich das Gesamtproblem zu lösen.

Da Prozesse keinen Speicher(memory und file handles) Teilen ist meist eine Kommunikation zwischen Prozessen notwendig, die durch das Betriebssystem bereitgestellt wird.

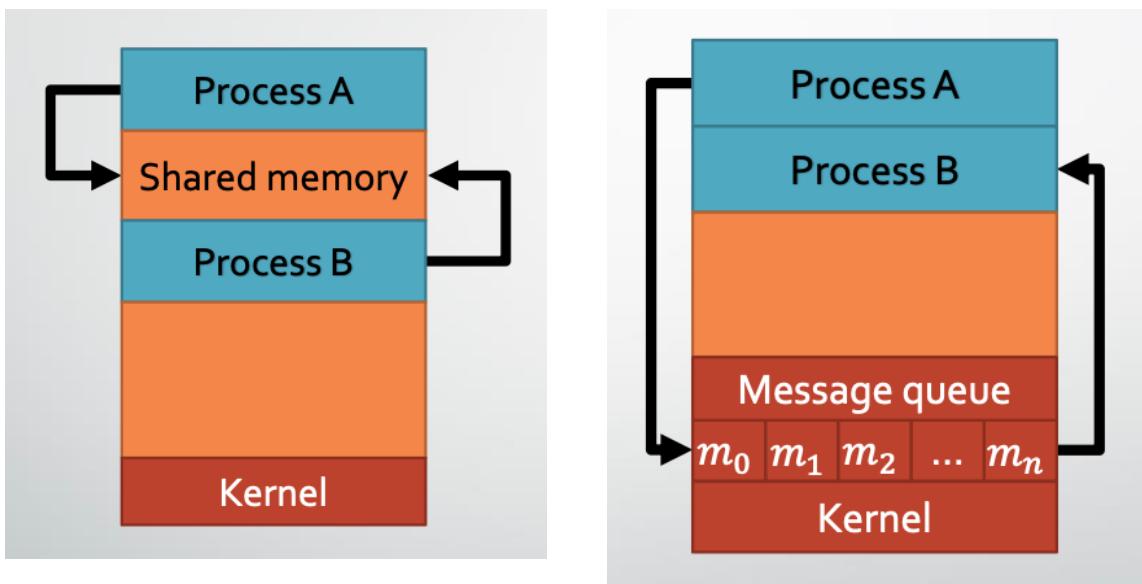
Zu dem bereits angesprochenen Vorteil der schnelleren Berechnung kommt durch das Aufteilen der Probleme auf unterschiedliche Programme/Prozesse eine verbesserte Modularität und somit auch eine bessere Fehlertoleranz bzw. Fehlerlokalität.

Ein *Nachteil* der Aufteilung ist die vermehrt benötigte Synchronisation, auf die in späteren Kapiteln eingegangen wird.

Beispiele für die Interprozesskommunikation sind:

- Dateien
- Pipes
- Shared Memory
- Signals
- Message Queues
- Sockets
- ...

13.1 Fundamentales Model



13.2 Messagepassing

- `send()`
- `receive()`

Erstellen einer IPC und Austausch von Daten über `send()` und `receive()`. Schlussendliches Löschen der IPC. Da die IPC mit mehreren Prozessen geteilt werden soll muss diese entweder im Gemeinsamen Speicher gespeichert werden, oder vor der Erstellung der Kindprozesse angelegt werden. Das Messagepassing kann bi- bzw unidirektional sein.

Synchronisation:

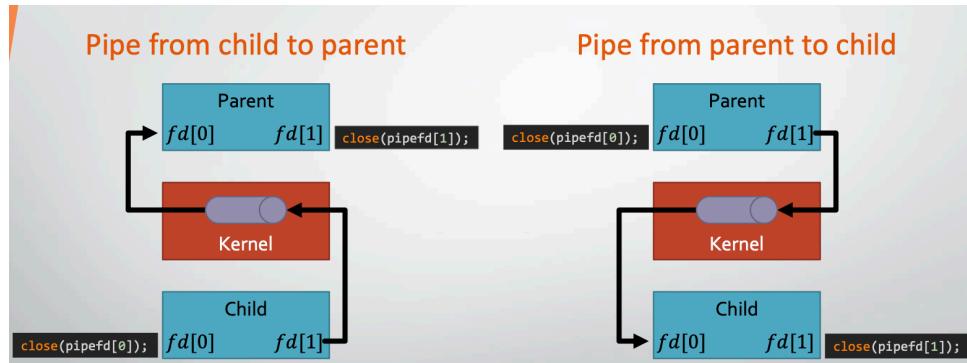
- Synchroner Austausch: blocking sender wartet bis Nachricht empfangen wurde, blocking receiver wartet bis eine Nachricht verfügbar ist.
- Asynchroner Austausch: non blocking sender sendet Nachricht und setzt fort, non blocking receiver empfängt eine Nachricht oder auch keine Nachricht.
- Rendezvous Kommunikation: blocking sender und receiver
- Hybrid: Synchron und Asynchron

Eine einfache IPC kann mit einer Datei implementiert werden. Dies ist jedoch extrem langsam und ineffizient. Die Synchronisation ist zudem nicht trivial und definitiv benötigt.

13.3 Unnamed Pipe

Unnamed Pipes werden von allen UNIX Systemen unterstützt. Sie sind unidirektional, haben also ein Read- und ein Write-end. Die Benutzung ist ähnlich zu den read/write Systemcalls. Daten können nur einmal gelesen werden und können jegliche Form haben. Der Kernel ist zuständig für die nötige Synchronisation.

Um eine Pipe zu erstellen muss der pipe Systemcall vor dem erstellen des Kind/Arbeitsprozesses aufgerufen werden. Passiert das nicht verbindet die Pipe zu dem gerade laufenden Prozess. Nach dem fork Aufruf sind Eltern- und Kindprozess mit der pipe verbunden.



```

void parent(const int pipefd[2]) {
    close(pipefd[0]);                                // Close read-end

    const char* msg = "Hello World!";
    write(pipefd[1], msg, strlen(msg));
    close(pipefd[1]);                                // Close write-end
                                                       // -> reader will see EOF

    wait(NULL);                                     // Wait for child
}

void child(const int pipefd[2]) {
    close(pipefd[1]);                                // Close write-end
    char buf;
    while(read(pipefd[0], &buf, 1) > 0) {
        write(STDOUT_FILENO, &buf, 1);
    }
    write(STDOUT_FILENO, "\n", 1);
    close(pipefd[0]);                                // Close read-end too
}

int main(void) {
    int pipefd[2];                                    // pipefd[0]: read-end
                                                       // pipefd[1]: write-end
    if(pipe(pipefd) != 0) return EXIT_FAILURE;

    const pid_t cpid = fork();
    if(cpid == -1) return EXIT_FAILURE;

    if(cpid == 0) child(pipefd);
    else          parent(pipefd);
}

```

13.4 Named Pipe (FIFO)

Named Pipes oder auch FIFO genannt werden von POSIX unterstützt und funktionieren ähnlich zu normalen Dateien. Sie können geöffnet, geschlossen, gelesen und beschrieben werden. Anders zu wirklichen Dateien werden die Daten jedoch nicht auf dem Sekundärspicher gespeichert. Sie werden vom Kernel verwaltet. Die Daten der FIFO können nur 1 mal gelesen werden. Im Gegensatz zu Unnamed Pipes können Named Pipes von mehreren Prozessen geöffnet werden. Eine Kommunikation ist jedoch erst möglich wenn beide Enden (Read/Write) geöffnet wurden. Wenn nur 1 Prozess schreibt und 1 Prozess liest ist die Synchronisation durch den Kernel sichergestellt.

```
void create_named_pipe_reader(void) {
    const char* name = "named_pipe";

    const mode_t permission = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH; // 644
    if(mkfifo(name, permission) != 0) return EXIT_FAILURE;

    const int fd = open(name, O_RDONLY);
    if(fd < 0) return EXIT_FAILURE;

    char buf;
    while(read(fd, &buf, 1) > 0) {
        write(STDOUT_FILENO, &buf, 1);
    }
    write(STDOUT_FILENO, "\n", 1);

    close(fd);
    unlink(name);
}

void create_named_pipe_writer(void) {
    const char* name ="named_pipe";
    const int fd = open(name, O_WRONLY);
    if(fd < 0) return EXIT_FAILURE;

    const char* msg = "Hello World";
    write(fd, msg, strlen(msg));

    close(fd);
}
```

13.5 Messagequeue

Messagequeues ermöglichen im Vergleich zu Pipes das Senden und Empfangen von Daten mit einer festgelegten Größe. Diese Größe wird gesendet UND empfangen und wird meist gehard-coded. Diese Pakete werden in der selben Reihenfolge empfangen wie sie gesendet wurden.

POSIX Messagequeues erlauben zu dem senden und empfangen die mitgabe der Priorität eines Pakets. Eine Höhere Priorität bedeutet das frühere Zustellen beim Empfänger, wohingegen die Reihenfolge von Paketen mit der selben Priorität erhalten bleibt.

- mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);

Erstellt oder Öffnet eine Messagequeue. Der Name (Konvention: Start mit "/") identifiziert die Queue aber ist nicht im Dateisystem sichtbar. Die oflag ist zuständig für den Zugriffstypen und ob die Queue erstellt werden darf.

mode sind die Berechtigungen der Queue.

attr legt die Eigenschaften der Messagequeue fest.

```

struct mq_attr {
    long mq_flags;           // queue flags, ignored on open
    long mq_maxmsg;          // max number of messages in the queue at any point
    long mq_msgsize;         // max size of each individual message
    long mq_curmsgs;         // number of messages in the queue
};

• int mq_send(mqd_t mqdes, const char* msg_ptr, size_t msg_len, unsigned int msg_prio);

Sendet eine Nachricht in die Messagequeue, mit den Daten msg_ptr und der Länge von
msq_size. msg_prio legt dir Priorität der Message fest.

• ssize_t mq_receive(mqd_t mqdes, char* msg_ptr, size_t msg_len, unsigned int* msg_prio);

Empfängt eine Nachricht aus der Messagequeue. Die Daten werden in msg_ptr mit der Länge
msq_size gespeichert. msg_prio ist die Priorität der empfangenen Message.

struct message {
    char data[32];
    bool quit;
};

typedef struct message message;

bool create_message_queue(const char* name) {
    const int oflag = O_CREAT | O_EXCL;
    const mode_t permissions = S_IRUSR | S_IWUSR; // 600
    const struct mq_attr attr = { .mq_maxmsg = 2, .mq_msgsize = sizeof(message) };
    const mqd_t mq = mq_open(name, oflag, permissions, &attr);
    if(mq == -1) return false;
    mq_close(mq);
    return true;
}

void logging_server(char **msg_queue_name) {
    const mqd_t mq = mq_open(*msg_queue_name, O_RDONLY, 0, NULL);
    for(int quit_requests = 0; quit_requests < 2;) {
        usleep(100 * 1000); // Simulate logging being very slow
        message msg = { 0 };
        unsigned int priority = 0;
        if(mq_receive(mq, (char*)&msg, sizeof(msg), &priority) == -1) return;

        if(msg.quit) ++quit_requests;
        else printf("%02u: %s\n", priority, msg.data);
    }
    mq_close(mq);
    mq_unlink(*msg_queue_name);
}

void client(char **msg_queue_name, long priority) {
    const mqd_t mq = mq_open(*msg_queue_name, O_WRONLY, 0, NULL);
    for(int i = 0; i < 10; ++i) {
        message msg = { .quit = false };
        sprintf(msg.data, "Hello World %d", i);
        if(mq_send(mq, (const char*)&msg, sizeof(msg), priority) != 0) return;
    }
    const message msg = { .quit = true };
    if(mq_send(mq, (const char*)&msg, sizeof(msg), priority) != 0) return;
    mq_close(mq);
}

int main(void) {
    pid_t cpid = fork();
    if(cpid == 0) {
        cpid = fork();
    }
}

```

```

if(cpid == 0) client(msg_queue_name, 0);
else         client(msg_queue_name, 1);
} else {
    logging_server(msg_queue_name);
}
}

```

13.6 Shared Memory

- `int shm_open(const char *name, int oflag, mode_t mode);`

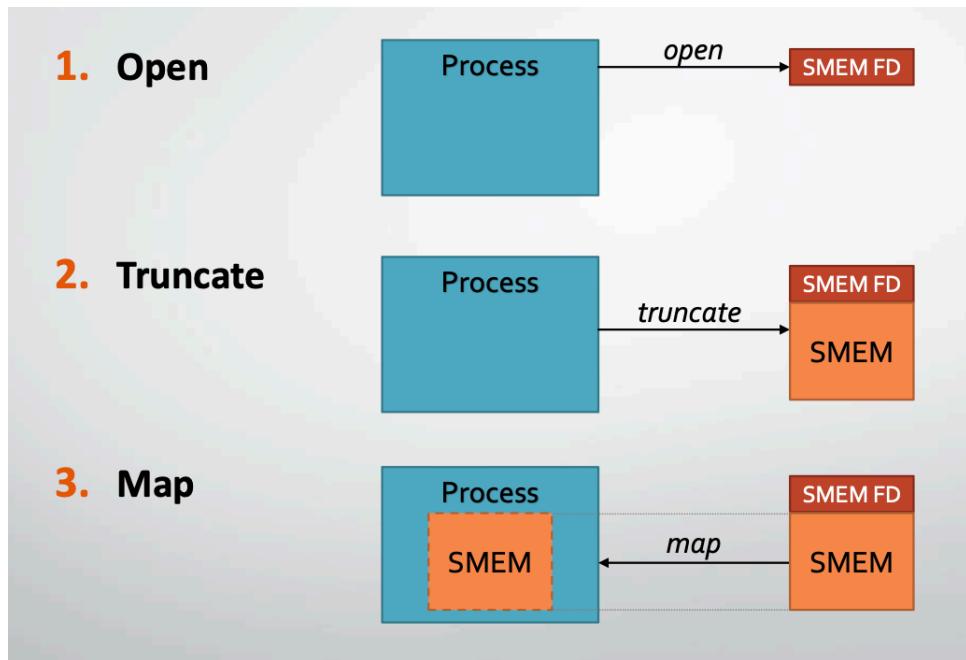
Erstellt oder öffnet ein Shared memory Objekt spezifiziert durch den Namen (Konvention: Start mit "/"). Dieses ist wie bei der Messagequeue nicht im Dateisystem zu finden. Wichtig ist hier das kein Speicher wirklich alloziert wird!

- `int ftruncate(int fd, off_t length);`

Begrenzt (oder erweitert) eine Datei auf die spezifizierte Größe. Hier ist wichtig das ftruncate nur auf allozierte Speicherbereiche angewendet werden darf.

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`

Dieser Befehl mapped Dateien oder Geräte in den Prozessspeicher. mmap wird benötigt um das shared_mem Objekt in den wirklichen Speicher zu legen. addr = NULL lässt den Kernel entscheiden wo genau das Memory Objekt alloziert/gemapped werden soll. length beschreibt die Größe des shared_mem Objekts. prot legt die Berechtigungen für den Speicherbereich fest. flags sollte bei shared_mem = MAP_SHARED. fd ist der Filedescriptor für das shared_mem Objekt. offset spezifiziert ob das Speicherobjekt mit einem gewissen offset gemapped werden soll. Bei einem fehlgeschlagenen Mapping wird MAP_FAILED zurückgegeben, sonst ein alloziert Speicherbereich.



```

void writer(void) {
    const char* name = "/shared_memory";
    const int oflag = O_CREAT | O_EXCL | O_RDWR; // create, fail if exists, read+write
    const mode_t permission = S_IRUSR | S_IWUSR; // 600
    const int fd = shm_open(name, oflag, permission);
    if(fd < 0) return EXIT_FAILURE;
}

```

```

const size_t shared_mem_size = 100;
if(ftruncate(fd, shared_mem_size) != 0) return EXIT_FAILURE;

char* shared_mem = mmap(NULL, shared_mem_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if(shared_mem == MAP_FAILED) return EXIT_FAILURE;

const char message[] = "Hello World";
memcpy(shared_mem, message, sizeof(message));

usleep(10 * 1000 * 1000);

munmap(shared_mem, shared_mem_size);
close(fd);
shm_unlink(name);
}

void reader(void) {
    const char* name = "/shared_memory";
    const int oflag = O_RDWR; // open read+write
    const int fd = shm_open(name, oflag, 0);
    if(fd < 0) return EXIT_FAILURE;

    const size_t shared_mem_size = 100;
    char* shared_mem = mmap(NULL, shared_mem_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    if(shared_mem == MAP_FAILED) return EXIT_FAILURE;

    char buffer[shared_mem_size];
    memcpy(buffer, shared_mem, shared_mem_size);

    munmap(shared_mem, shared_mem_size);
    close(fd);

    printf("%.*s\n", (int)shared_mem_size, buffer);
}

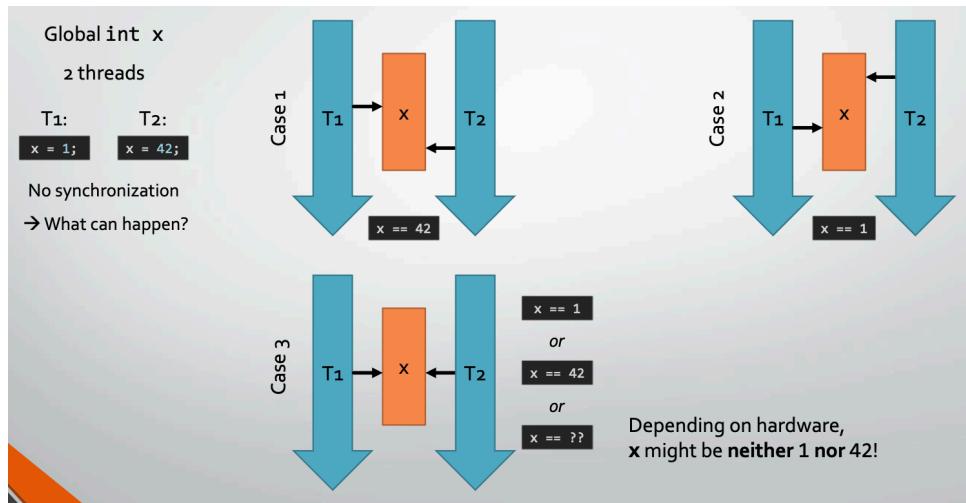
```

Beim Shared Memory stellt sich nun jedoch ein Problem. Wann weiß das Lesende Ende wann Daten verfügbar sind? Was passiert wenn gleichzeitig gelesen **und** geschrieben wird? Was ist wenn das Lesende Ende Daten zurücksenden will?

Hier wird eine Art von Synchronisation benötigt. Der usleep call ist nicht ausreichend.

14 Synchronisation

Wir benötigen Synchronisation, da Prozesse und Threads gleichzeitig Daten verarbeiten. Wenn nun mehrere Prozesse wieder gleichzeitig auf die selben Daten zugreifen kann es zu Fehlern kommen. Diese Fehler werden **race conditions** genannt.



Oder in Code:

```
#define ITERATIONS 100000000
#define THREADS 2
int x = 0;
void* thread(void* param) {
    for(int i = 0; i < ITERATIONS; ++i) {
        ++x;
    }
    return NULL;
}
int main(void) {
    /* spawn THREADS number of threads and wait until they finish */
    printf("x: %d", x);
}
```

14.1 Data Hazards

- **Read-after-Write (RAW):** true dependency

- 1) $x = 4;$
- 2) $y = x + 7;$
- 3) $x = 2;$

y hängt vom Wert der Variable x ab. 1 und 2 haben eine true dependency. Wird Zeile 1 und 2 getauscht ist der Output ein anderer.

- **Write-after-Read (WAR):** anti dependency

- 1) $x = 4;$
- 2) $y = x + 7;$
- 3) $x = 2;$

y hängt vom Wert der Variable von x ab. x wird später verändert. Wenn nun Zeile 2 und 3 getauscht werden ist der Output ein anderer.

- **Write-after-Write (WAW):** output dependency

- 1) $x = 4;$
- 2) $y = x + 7;$
- 3) $x = 2;$

y hängt vom Wert der Variable von x ab. x wird in Zeile 1 und 3 verändert. Wenn nun Zeile 1 und 3 getauscht werden ist der Output ein anderer.

14.2 Critical Sections

In dem oben gezeigten Code Abschnitt ist `++x` die Critical section. Jede Critical Section benötigt eine Synchronisation bei einer parallelen Ausführung.

14.3 Atomics

Atomics sind der Grundstein für die Synchronisation. Sie benötigten Hardware support und können dementsprechend entweder komplett funktionieren oder gar nicht. Mit Atomics können wir die oben angesprochenen Probleme vollständig lösen.

`<stdatomic.h>` stellt primitive Datentypen wie `atomic_bool` oder `atomic_int` zur Verfügung. Benutzerdefinierte Typen können mit `_Atomic` deklariert werden und sind demnach `Atomic`. Wird `_Atomic` verwendet müssen die `Atomic API` Funktionen verwendet werden.

Das oben gegebene Beispiel kann durch die Benutzung eines `atomic_int` als Datentyp für `x` synchronisiert werden, da der parallele Zugriff auf Atomics sicher ist.

14.4 Atomic Operationen

Operation	Explanation
<code>atomic_store</code>	stores a value in an atomic object
<code>atomic_load</code>	reads a value from an atomic object
<code>atomic_exchange</code>	swaps a value with the value of an atomic object
<code>atomic_compare_exchange_*</code>	swaps a value with an atomic object if the old value is what is expected, otherwise reads the old value
<code>atomic_fetch_add</code>	atomic addition
<code>atomic_fetch_sub</code>	atomic subtraction
<code>atomic_fetch_or</code>	atomic bitwise OR
<code>atomic_fetch_xor</code>	atomic bitwise exclusive OR
<code>atomic_fetch_and</code>	atomic bitwise AND

14.5 Shared Memory mit Synchronisation

```
struct shared_data {
    atomic_bool available;
    atomic_bool processed;
    size_t cnt;
    char buf[1024];
};

typedef struct shared_data shared_data;
```

Dieses Konstrukt stehen dem Schreibenden **und** dem Lesenden Ende zur Verfügung. Der Server:

```
/* shared memory creation and mapping, same as before */
data->processed = true; // Indicate initialization is done
while(!data->available); // Busy wait for data to become available
// Process data
for(size_t i = 0; i < data->cnt; ++i) {
    data->buf[i] = toupper((unsigned char) data->buf[i]);
}
// Signal data has been processed
data->processed = true;
munmap(data, sizeof(shared_data));
close(fd);
shm_unlink(shared_mem_name);
```

Der Client:

```

/* shared memory creation and mapping, same as before */
while(!data->processed); // Busy wait until shared memory has been initialized
data->processed = false; // Reset flag
// Write data
data->cnt = strlen(message);
memcpy(&data->buf, message, data->cnt);
// Signal data has been written
data->available = true;
while(!data->processed); // Busy wait until data has been processed
printf("%.*s\n", (int)data->cnt, data->buf);
munmap(data, sizeof(shared_data));
close(fd);

```

14.6 Mutex

Mutex (von **Mutual Exclusion**), auch genannt Locks können 2 States halten:

- Locked/held/owned/acquired
- Unlocked/free/released.

Ein Mutex kann immer nur von **einem** Thread gehalten/gelocked sein. Jeder weitere Thread der versucht den Mutex zu halten wartet bis dieser erneut freigegeben wird. Die Nutzung eines Mutex bringt jedoch einen großen Performance Verlust mit sich, weshalb die Critical Section klein gehalten werden sollte.

Mutexes werden meist durch eine Binäre Semaphore dargestellt (siehe Semaphores 14.16).

Das obige Problem, zuerst gelöst durch die Verwendung von Atomics kann auch mit der Verwendung eines Mutex gelöst werden:

```

pthread_mutex_t mutex;
int x = 0;
void* thread(void* param) {
    for(int i = 0; i < ITERATIONS; ++i) {
        pthread_mutex_lock(&mutex);
        ++x;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
• int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* mutexattr);

```

oder

- `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`

erstellt den Mutex mit den spezifizierten mutexattr bzw. einen default Mutex wenn mutexattr = NULL.

- `int pthread_mutex_lock(pthread_mutex_t* mutex);`

versucht den Mutex zu locken. Ist dieser nicht gelocked, locked der aufrufende Thread den Mutex sofort. Sonst blockt der Thread so lange bis der Mutex unlocked ist.

- `int pthread_mutex_trylock(pthread_mutex_t* mutex);`

versucht den Mutex zu locken. Ist dieser nich gelocked, locked der aufrufende Thread den Mutex sofort. Sonst wird der error code EBUSY zurückgegeben.

Das Locken des Mutex darf im Thread nur einmal passieren.

- `int pthread_mutex_unlock(pthread_mutex_t* mutex);`

unlocked den Mutex. Der Mutex muss zuvor von den Thread gelocked worden sein.

- `int pthread_mutex_destroy(pthread_mutex_t* mutex);`

zerstört den Mutex nach der Benutzung. Der Mutex darf zu diesem Zeitpunkt nicht mehr gehalten werden.

14.7 Dekker's Algorithmus

(1960er)

Hardware support für Mutexes hat es noch nicht immer gegeben und der Dekker Algrotihmus war die erste korrekte Lösung für das Mutex Problem. Der Dekker Algorithmus ist in Software implementiert, ohne jeglichen Hardware support. Dieser arbeitet auf dem Gemeinsamen Speicher. Auf neuen Systemen nicht funktionsfähig und nicht möglich rein in C zu lösen, da der Compiler und die CPU heutzutage in der Lage sind Instruktionen umzustellen wenn der Code nicht synchronisiert ist. Dies verhindert das Benutzen des Dekker Algorithmus.

Dekker Algorithmus in Pesudocode:

```
variables:  
    wants_to_enter: array of 2 booleans  
    turn: integer  
    wants_to_enter[0] ← false  
    wants_to_enter[1] ← false  
    turn ← 0 // or 1  
  
p0:  
    wants_to_enter[0] ← true  
    while wants_to_enter[1] {  
        wants_to_enter[0] ← false  
        while turn != 0 // busy wait  
        wants_to_enter[0] ← true  
    }  
    // critical section  
    turn ← 1  
    wants_to_enter[0] ← false  
    // remainder section  
  
p1:  
    wants_to_enter[1] ← true  
    while wants_to_enter[0] {  
        wants_to_enter[1] ← false  
        while turn != 1 // busy wait  
        wants_to_enter[1] ← true  
    }  
    // critical section  
    turn ← 0  
    wants_to_enter[1] ← false  
    // remainder section
```

14.8 Peterson's Algorithmus

(1981)

Der Peterson's Algorithmus ist wie der Dekker Algorithmus eine Software Lösung für das Mutex Problem. Der Peterson's Algorithmus wird als besser angesehen, da er einfacher zu implementieren ist, weniger Anweisungen und eine klarere Logik besitzt und besser für formale Beweise geeignet ist.

Peterson's Algorithmus in Pseudocode:

```
variables:  
    flag: array of 2 booleans  
    turn: integer
```

```

flag[0] ← false
flag[1] ← false

p0:
    flag[0] ← true
    turn ← 1
    while (flag[1] && turn == 1)
        // critical section
        flag[0] ← false
        // remainder section

p1:
    flag[1] ← true
    turn ← 0
    while (flag[0] && turn == 0)
        // critical section
        flag[1] ← false
        // remainder section

```

Wie auch bereits beim Dekker Algorithmus angesprochen ist das benutzen dieses Algorithmus heutzutage nicht mehr möglich, wegen Compiler und CPU Instruction reordering.

14.9 Memorybarrier

Heutige CPUs optimieren die Ausführung stark. Um schneller zu arbeiten werden häufig Befehl umgeordnet und verzögern Speicheroperationen. In Multi-Core-Systemen können dadurch Speicheränderungen nicht sofort für andere Threads sichtbar sein. Selbst bei korrektem Code, wie die beiden oben angesprochenen Algorithmus, kann es zu race conditions kommen.

Wir Unterscheiden zwischen:

- stark geordnetem Speicher (Änderungen sind sofort für alle Prozessoren sichtbar)
- und
- schwach geordnetem Speicher (Änderungen sind nicht sofort sichtbar. Probleme bei der Parallelität).

Thread 1	Thread 2
-----	-----
while(!flag)	turn = 1;
memory_barrier()	memory_barrier();
print turn;	flag = true;

Hier benutzen wir eine Speicherbarriere auf schwach geordneten Speicher Systemen um sicherzustellen, dass das Speichern in der richtigen Reihenfolge stattfindet. Ohne Barriere könnte die CPU die Instruktionen umordnen und es kommt zu einer race condition.

14.10 Deadlocks

2 Thread halten bzw. geben 2 Mutexes frei. Dies geschieht in entgegengesetzter Reihenfolge. Dies kann nicht funktionieren da nun beide Threads auf die Freigabe des jeweilig anderen Mutex wartet, diese aber erst freigegeben werden, wenn einer der beiden Mutexes freigegeben wird. Dies ist in sich ein Widerspruch und somit unmöglich.

Beispiel in C:

p0:

p1:

```

for(int i = 0; i < ITERATIONS; ++i) {
    pthread_mutex_lock(&mutexes[0]);
    // hält hier
    // nächstes locking unmöglich
    pthread_mutex_lock(&mutexes[1]);

    pthread_mutex_unlock(&mutexes[1]);
    pthread_mutex_unlock(&mutexes[0]);
}
}

for(int i = 0; i < ITERATIONS; ++i) {
    pthread_mutex_lock(&mutexes[1]);
    // und hier (gleichzeitig)
    // nächstes locking unmöglich
    pthread_mutex_lock(&mutexes[0]);

    pthread_mutex_unlock(&mutexes[0]);
    pthread_mutex_unlock(&mutexes[1]);
}
}

```

Eine weitere Art von Deadlock ist das Zirkuläre Warten. Dies geschieht wenn eine begrenzte Anzahl an Threads auf das Lock des jeweilig nächsten wartet und sich dadurch ein Zyklus bildet. Es ist unmöglich für einen der Threads fort zu fahren. Dieses Problem gleicht dem Dining Philosophers Problem.

Deadlock Prävention:

- Hold and Wait: Alle Ressourcen am Anfang anfordern und erst am Schluss wieder freigeben. Insgesamt jedoch eine geringe Ressourcennutzung.
- No preemption: Freigeben aller Ressourcen wenn benötigte Ressourcen nicht angefordert werden können. Ausführung fortfahren wenn alle benötigten Ressourcen angefordert werden können.
- Circular Waiting: Ressourcen in aufsteigender Reihenfolge anfordern.

Hier verweise ich erneut auf die Folien von Herr Radush (*Foliensatz 5, Folien 51 bis 56*), da ich keine Ahnung habe was wirklich passiert!

14.11 Bankieralgorithmus

Wir haben n Threads und m Ressourcentypen:

- einen Ressource-Availability Vector: avail_m
- eine Max-Acquire Matrix: max_{nm}
- eine Ressource-Map Matrix: alloc_{nm}
- eine Ressources-Needed Matrix: need_{nm}

Safety Algorithm:

Input: $n, m, \text{avail}[n][m], \text{alloc}[n][m], \text{need}[n][m]$
Output: True/False (safe or unsafe)

```

finish[1..n] = false

i = 1
while i <= n && finish[i] == false && need[i] <= avail:
    finish[i] = true
    avail = avail + alloc[i]

return finish[1] && ... && finish[n]
⇒ O(n² · m)

```

Ressourcesneeded Algorithm:

Input: $n, m, \text{avail}[n][m], \text{alloc}[n][m], \text{need}[n][m], \text{req}[i]$ // resources need by Thread T[i]
Output: True/False (safe or unsafe)

```

if req[i] <= need[i]
    error: needs too many Ressources

if req[i] > avail:
    wait(T[i])
else:

```

```

avail = avail - req[i]
alloc[i] = alloc[i] + req[i]
need[i] = need[i] - req[i]
if !safe(n, m, avail, alloc, need):
    avail = avail + req[i]
    alloc[i] = alloc[i] - req[i]
    need[i] = need[i] + req[i]

```

Beispiel:

- Sichere Threadreihenfolge:

$$\{T_1, T_3, T_4, T_2, T_0\}$$

- $\text{Req}_1 = (1, 0, 2) < (3, 3, 2) = \text{Avail}$
 - $\text{Avail} = (3, 3, 2) - (1, 0, 2) = (2, 3, 0)$
 - $\text{Alloc}_1 = (3, 0, 2), \text{Need}_1 = (0, 2, 0)$
- Sichere Threadreihenfolge:

$$\{T_1, T_3, T_4, T_0, T_2\}$$

- $\text{Req}_4 = (3, 3, 0) > (2, 3, 0) = \text{Avail}$
 - Anforderung größer als Available
- $\text{Req}_0 = (0, 3, 0) < (2, 3, 0) = \text{Avail}$
 - $\text{Need}_0 = (7, 1, 3), \text{Alloc}_0 = (0, 4, 0)$
 - Unmöglich da unsicherer Zustand

Avail	Thread	Alloc			Need		
		A	B	C	A	B	C
A	B	C					
3	T_0	0	1	0	7	4	3
3	T_1	2	0	0	1	2	2
3	T_2	3	0	2	6	0	0
2	T_3	2	1	1	0	1	1
0	T_4	0	0	2	4	3	1

Avail	Thread	Alloc			Need		
		A	B	C	A	B	C
A	B	C					
2	T_0	0	1	0	7	4	3
3	T_1	3	0	2	0	2	0
3	T_2	3	0	2	6	0	0
2	T_3	2	1	1	0	1	1
0	T_4	0	0	2	4	3	1

14.12 Deadlock Detection

Immer nur eine Instanz pro Ressourcentyp. Darstellung durch einen Resourczuteilungsgraph, einem Wartegraph und einem Zykluserkennungsalgorithmus.

Cycledetection Algorithm:

Input: $n, m, \text{avail}[n][m], \text{alloc}[n][m], \text{req}[n][m]$
 Output: True/False (safe or unsafe)

```

finish[i] = alloc[i] != 0 ? False : True; i = {1..n}

i = 0
while i <= n && finish[i] = False && req[i] <= avail
    finish[i] = True
    avail = avail + alloc[i]

return finish[1] && ... && finish[n]
 $\Rightarrow \mathcal{O}(n^2 \cdot m)$ 

```

Beispiel:

- Sicherer Zustand:
 - Kein Deadlock

$\{T_0, T_2, T_3, T_1, T_4\}$

\Rightarrow

Thread	Alloc			Req		
	A	B	C	A	B	C
T_0	0	1	0	0	0	0
T_1	2	0	0	2	0	2
T_2	3	0	2	0	0	0
T_3	2	1	1	1	0	0
T_4	0	0	2	0	0	2

- Unsicherer Zustand:
 - Deadlock

$\{T_0, \dots\}$

\Rightarrow

Thread	Alloc			Req		
	A	B	C	A	B	C
T_0	0	1	0	0	0	0
T_1	2	0	0	2	0	2
T_2	3	0	2	0	0	1
T_3	2	1	1	1	0	0
T_4	0	0	2	0	0	2

14.13 Deadlock Behebung

Um einen Deadlock zu beheben können entweder alle Threads die in dem Deadlock gefangen sind gleichzeitig oder nacheinander terminiert werden. Beim sequenziellen Terminieren wird nach jeder Terminierung überprüft ob das System nun Deadlock frei ist. Die Reihenfolge der Terminierung kann von Threadpriorität, Berechnungszeit und weiteren Faktoren abhängen.

Eine weitere Möglichkeit ist die Präemption von Ressourcen. Hierbei werden Ressourcen die in einem Deadlock gefangen sind Ressourcen entzogen, bzw. einzelne Threads gezielt terminiert.

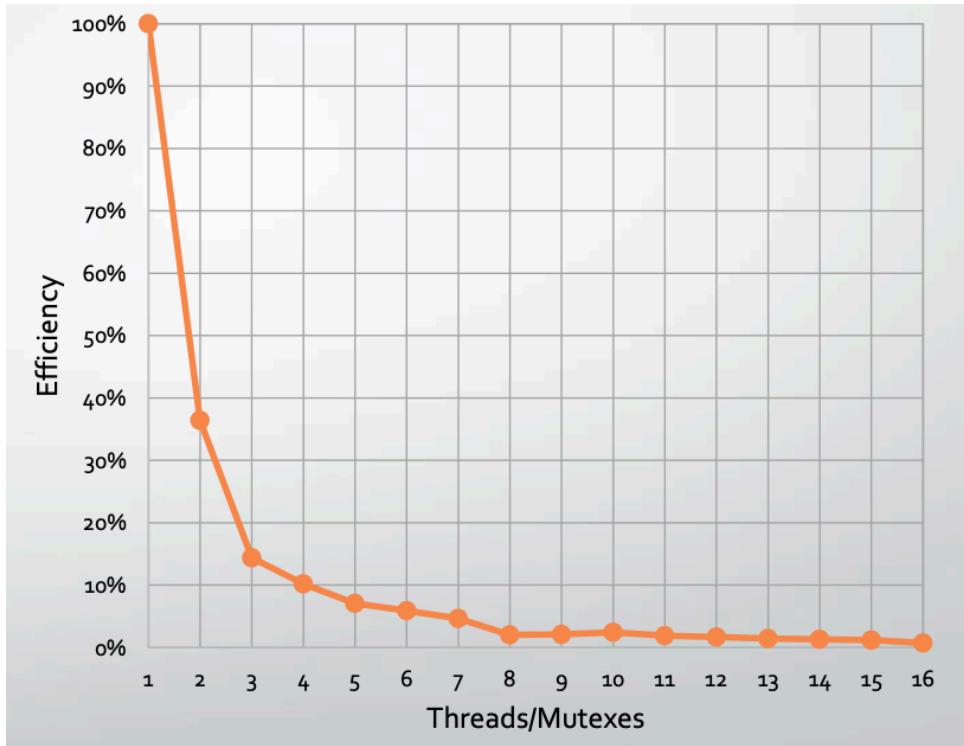
Außerdem können Threads auf einen früheren Zustand zurückgesetzt bzw. neu gestartet werden.

14.14 Livelocks

```
void* thread(void* param) {
    ThreadData* data = param;
    for(int i = 0; i < ITERATIONS; ++i) {
        if(!trylock_all_mutexes(data)) continue;
        ++data->work_done;
        unlock_all_mutexes(data);
    }
    return NULL;
}
```

N Threads versuchen N mutexes zu locken um eine bestimmte Aufgabe zu lösen. Dies ist ähnlich zu einem Deadlock. Die Ausführung des Programms stoppt nicht. Die CPU verbraucht jedoch Ressourcen ohne wirkliche Arbeit zu leisten.

Effizienz:



Effizienz wird als Arbeit definiert die wirklich geschafft wird. Nicht als Versuche Arbeit zu leisten. Wenn hier nun die Anzahl der Threads und somit die Anzahl der Mutexes steigt, nimmt die Effizienz drastisch ab. Durch das zurückziehen finden keinen wirklichen Deadlocks statt.

14.15 Condition Variables

Condition Variables werden genutzt um zu warten/blockieren bis eine bestimmte Kondition wahr wird. Sie können ein Signal erhalten durch das signalisiert wird das eine Bestimmte Kondition erfüllt ist. Die meisten praktischen Anwendung haben aber sogannente *spurious wakeups*. Das bedeutet das die Condition Variable ein Signal bekommt ohne das die Kondition wirklich erfüllt ist. Diese müssen explizit behandelt werden.

- `int pthread_cond_init(pthread_cond_t* cond, pthread_condattr_t* cond_attr);`

initialisiert die Condition Variable, die durch den pointer gegeben wird. cond_attr spezifiziert die Attribute der Condition Variable.

- `int pthread_cond_destroy(pthread_cond_t* cond);`

zerstört die Condition Variable.

- `int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);`

jede Condition Variable muss mit einem Mutex genutzt werden. Der oben genannte Befehl muss auf einen bereits gehaltenen Mutex folgen. Der Aufruf überprüft ob die Condition Variable ein Signal erhält. Ist dies nicht der Fall bzw. ein *spurious wakeup*-call findet statt, wird der Mutex freigegeben und der Aufruf blockiert die Ausführung. Bekommt die Condition Variable ein Signal werden die folgenden Anweisungen ganz normal ausgeführt. Beim zurückkehren des Aufrufs ist der Mutex entweder immer noch gelocked(Signal erhalten) oder wird geredlocked(Blockieren und dann erst Signal).

Um *spurious wakeups* zu behandeln wird die Kondition nach dem zurückkehren der Funktion erneut überprüft. Dafür wird meistens eine while-Schleife verwendet. Dies bringt den Vorteil, dass bei einer erfüllten Kondition der Aufruf gar nicht erst stattfindet. Ein einfaches Beispiel ist:

```

pthread_mutex_lock(&mutex);
    while(!condition) {
        pthread_cond_wait(&cond, &mutex);
    }
// Critical section
pthread_mutex_unlock(&mutex);
• int pthread_cond_signal(pthread_cond_t* cond);

```

signalisiert genau einem Thread das die Kondition erfüllt ist. Dieser verlässt nun den pthread_cond_wait Aufruf. Warten keine Threads passiert nichts. Warten mehrere Threads verlässt ein unspezifizierter Thread den pthread_cond_wait Aufruf.

- int pthread_cond_broadcast(pthread_cond_t *cond);

signalisiert allen wartenden Threads, und veranlasst diese den pthread_cond_wait Aufruf zu verlassen. Da die Threads durch einen Mutex blocken folgt dies keiner spezifizierten Reihenfolge und die Threads verlassen nacheinander den Aufruf.

Beispiel:

client:

```

pthread_mutex_lock(&data->mtx);           // Aquire mutex
data->cnt = strlen(message);             // Write
memcpy(&data->buf, message, data->cnt); // message
data->available = true;                 // Set condition
pthread_mutex_unlock(&data->mtx);        // Release mutex
pthread_cond_signal(&data->cond);        // Signal condition variable

pthread_mutex_lock(&data->mtx);           // Aquire mutex
while(!data->processed) {                // Wait until condition
    pthread_cond_wait(&data->cond, &data->mtx); // Release mutex and wait
}                                         // Mutex will be re-aquired
printf("%.*s\n", (int)data->cnt, data->buf); // Print
pthread_mutex_unlock(&data->mtx);        // Release mutex

```

server:

```

pthread_mutex_lock(&data->mtx);           // Aquire mutex
while(!data->available) {                // Wait until condition
    pthread_cond_wait(&data->cond, &data->mtx); // Release mutex and wait
}
for(size_t i = 0; i < data->cnt; ++i) {   // Process
    data->buf[i] = toupper((unsigned char)data->buf[i]); // the
}                                         // data
data->processed = true;                  // Set condition
pthread_mutex_unlock(&data->mtx);        // Release mutex
pthread_cond_signal(&data->cond);        // Signal condition variable

```

14.16 Semaphores

Semaphores sind ähnlich zu Mutexes. Semaphoren sind im größten Sinne ein Zähler. Dieser Zähler startet meistens bei 0 und zeigt meistens an, ob Ressourcen verfügbar sind. Sie ermöglichen 2 Operationen.

- post/release: welches den Zähler um 1 erhöht und die Semaphore semantisch unlocked, und
- wait/acquire: welches den Zähler um 1 verringert und die Semaphore sematisch locked.

Das Verringern ist nur möglich wenn der Zähler > 0 ist. Eine Semaphore die zu jedem Zeitpunkt den Wert 0 bzw. 1 hält wird auch Binäre Semaphore genannt bzw. Mutex genannt.

- int sem_init(sem_t* sem, int pshared, unsigned int value);

initialisiert die Semaphore. Wenn pshared = 0 wird die Semaphore zwischen Threads geteilt. Ist pshared ≠ 0 dann wird die Semaphore zwischen Prozessen geteilt. Die Semaphore muss beim teilen mit Prozessen im Shared memory liegen. Das initialisieren muss nur einmal und nicht pro Thread/Prozess erfolgen. Value gibt den Startwert der Semaphore an.

- `int sem_destroy(sem_t* sem);`

zerstört die Semaphore. Muss nur einmal und nicht pro Thread/Prozess erfolgen.

- `int sem_wait(sem_t* sem);`

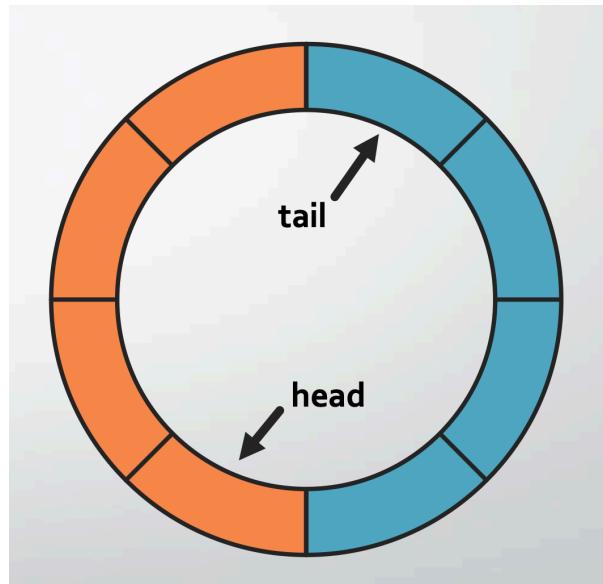
verringert den Wert der Semaphore. Ist der Wert der Semaphore 0, blockiert der Aufruf bis ein anderer Thread die Semaphore erhöht.

- `int sem_post(sem_t* sem);`

erhöht den Wert der Semaphore. War der Wert der Semaphore 0, wird ein blockender `sem_wait` call entblocked.

14.17 Ring Buffer

Ein Ring Buffer wird verwendet um eine Queue zu implementieren. Er hat ein read-end(tail) und ein write-end(head). Wenn head = tail ist der Ring Buffer leer. Wenn (head + 1) mod N = tail ist der Ring Buffer voll. Die Kapazität des Ring Buffers ist $N - 1$, da sonst "leer" und "voll" nicht klar definiert wäre.



```

struct RingBuffer {
    int head;
    int tail;
    Data data[BUFFER_SIZE];
};

typedef struct RingBuffer RingBuffer;

void ring_buffer_init(RingBuffer* buf) {
    buf->head = 0;
    buf->tail = 0;
}

bool ring_buffer_is_full(RingBuffer* buf) {
    return ((buf->head + 1) % BUFFER_SIZE) == buf->tail;
}

```

```

bool ring_buffer_is_empty(RingBuffer* buf) {
    return buf->head == buf->tail;
}

bool ring_buffer_push(RingBuffer* buf, const Data* data) {
    if(ring_buffer_is_full(buf)) return false;
    buf->data[buf->head] = *data;
    buf->head = (buf->head + 1) % BUFFER_SIZE;
    return true;
}

bool ring_buffer_pop(RingBuffer* buf, Data* data) {
    if(ring_buffer_is_empty(buf)) return false;
    *data = buf->data[buf->tail];
    buf->tail = (buf->tail + 1) % BUFFER_SIZE;
    return true;
}

```

14.18 Producer - Consumer Problem



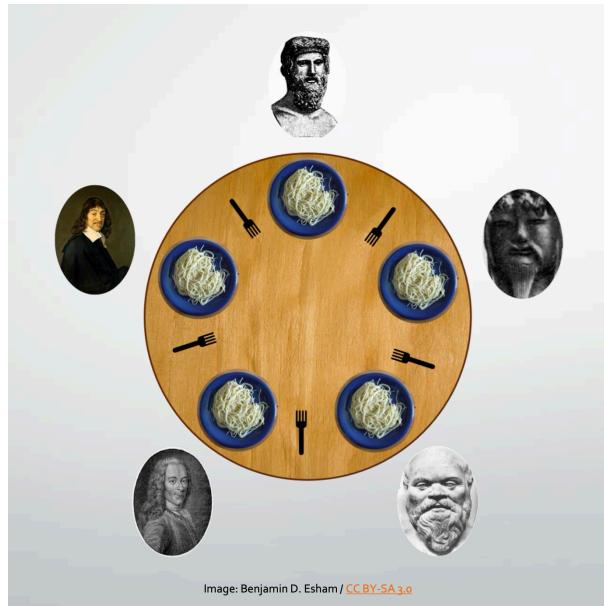
Das Producer-Consumer Problem ist ein klassisches Problem in der Informatik. Der Producer und die Consumer sind Threads bzw. Prozesse, die parallel arbeiten. Der Producer erzeugt ununterbrochen Daten und speichert diese in dem Ringbuffer. Die Consumer nehmen sich Daten aus dem Ring Buffer und verarbeiten diese. Auf die Frage “Wird hier Synchronisation benötigt” stellen wir uns ein Beispiel eines Buffets vor.

Auf diesem Buffet wird ständig neues Essen gekocht und gegessen. Der Koch fügt neues Essen hinzu wenn Platz auf dem Buffet ist bzw. die Gesamtanzahl der benötigten Essen noch nicht erschöpft ist. Jedoch ist diese Gesamtanzahl viel größer als Platz auf dem Buffet ist. Die Consumer bzw. Gäste essen, solange Essen auf dem Buffet ist bzw. die Gesamtanzahl der Essen konsumiert wurde. Wenn das Buffet leer ist wartet der Consumer auf das nächste Gericht.

Ohne Synchronisation greifen mehrere Gäste gleichzeitig auf das Buffet zu. Dabei können sie sich in die Quere kommen. Am Beispiel: Es befindet sich ein Essen auf dem Buffet. 2 oder mehr Gäste sehen dieses Essen und konsumieren dies (virtuell) gleichzeitig. Entweder die Gäste konsumieren Essen das eigentlich gar nicht da ist, oder sie *greifen* sich ein Essen das schon nicht mehr vorhanden ist. In beiden Fällen findet eine **race condition** statt bzw. potentiell ein Error statt.

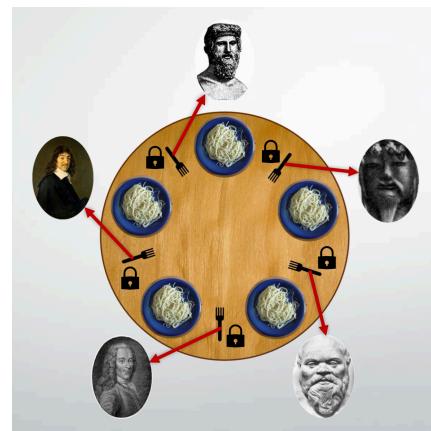
Das Code Beispiel zu diesem Problem kann im *Foliensatz 4, Folien 37-40* nachvollzogen werden. Um das Problem zu beheben verwenden wir 2 Semaphoren, die anzeigen wie viel freier Platz auf dem Buffet ist, und wie viele Essen sich auf dem Buffet befinden. Zudem benötigen wir einen Mutex der den Zugriff auf das Buffet schützt und einen Atomic Boolean der den Consumern angezeigt ob diese Stoppen sollen.

14.19 Dining Philosophers Problem



An einem Tisch befinden sich N Philosophen. Jeder Philosoph hat eine Portion Spaghetti vor sich. Auf dem Tisch befinden sich zudem $N - 1$ Essstäbchen. Ein Philosoph wechselt immer von Denken zu Essen. Um zu essen muss jeder Philosoph die 2 Essstäbchen neben seinen Spaghetti aufheben. Das Aufheben kann durch einen Mutex für jedes Essstäbchen synchronisiert werden.

Dies kann jedoch einfach zu einem Deadlock führen, wenn alle Philosophen gleichzeitig zum Beispiel das Stäbchen rechts seines Essens aufhebt. Nun ist für keinen der Philosophen mehr möglich ein weiteres Stäbchen aufzuheben und seine Spaghetti zu essen. Wir befinden uns in einem Deadlock!



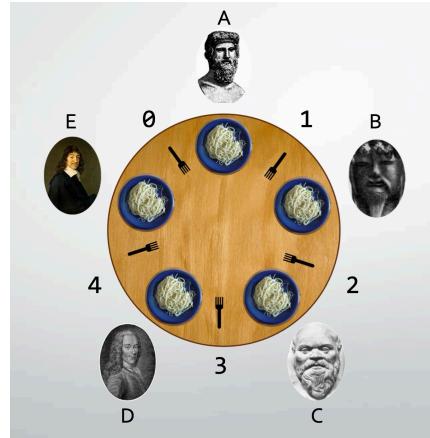
Lösungsansätze:

- **Centralized Locking:**

Es wird eine zentraler Mutex verwendet um die Kontrolle der Esstäbchen zu schützen. Jeder Philosoph kann den Mutex halten und versuchen die beiden Stäbchen links und rechts aufzuheben. Gelingt dies werden die Stäbchen aufgehoben und der Mutex wird freigegeben. Gelingt dies nicht werden die Stäbchen zurückgelegt und es wird zu einem späteren Zeitpunkt erneut versucht. Durch das warten kann es jedoch zu einem Livelock kommen und manche Philosophen essen nie und Verhungern.

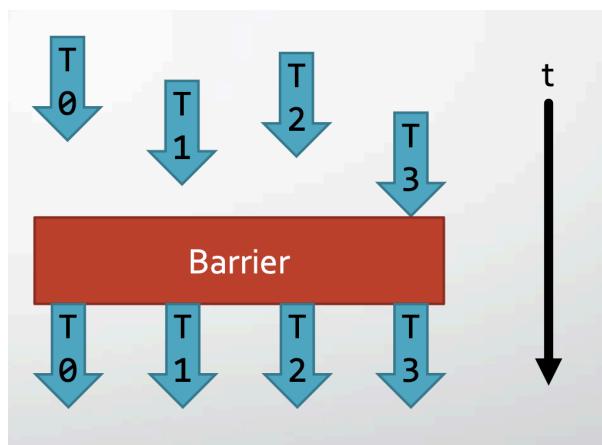
- **Ordering:**

Die Stäbchen bekommen eine Reihenfolge. Jeder Philosoph hebt erst das Stäbchen mit der niedrigeren ID auf. Hier kommt es jedoch dazu, dass manche Philosophen häufiger essen als andere. In dem Beispiel isst Philosoph häufiger als andere.



14.20 Barriers

Barriers synchronisieren alle partizipierenden Threads, indem sie die Ausführungen solange blockieren bis alle Threads an der Barriere angelangt sind.



- `int pthread_barrier_init(pthread_barrier_t* barrier,
const pthread_attrattr_t* attr,
unsigned count);`

initialisiert die Barrier mit den spezifizierten Attributen und dem count als Anzahl der partizipierenden Threads.

- `int pthread_barrier_destroy(pthread_barrier_t* barrier);`

zerstört die Barrier.

- `int pthread_barrier_wait(pthread_barrier_t* barrier);`

das Ankommen an der Barrier wird auch warten genannt. Der Aufruf blockt solange bis alle Threads an der Barriere angekommen sind. Ein unspezifizierte Thread gibt den Wert `PTHREAD_BARRIER_SERIAL_THREAD` zurück. Dies kann benutzt werden um bestimmte Logik pro Barrier auszuführen bzw. pro Loop/Step.

Beispiel:

```
void* thread(void* arg) {
    ThreadData* thread_data = arg;
    SharedData* data = thread_data->data;
    for(int t = 0; t < NUM_TIMESTEPS; ++t) {
        compute(thread_data);
        if(pthread_barrier_wait(&data->barrier) == PTHREAD_BARRIER_SERIAL_THREAD) {
            swap_buffers(data);
```

```

    }
    pthread_barrier_wait(&data->barrier);
}
return NULL;
}

```

14.21 Interrupt-based Synchronisation

Generelles Vorgehen:

1. Eintritt in Critical Section: Deaktivieren der Interrupts
2. Ausführen der Critical Section
3. Austritt aus Critical Section: Reaktivieren der Interrupts

Auf Single-Core-Systemen äußerst effizient und einfach zu implementieren. Sie garantiert den exklusiven Zugriff in einer Critical-Section. Auf Multi-Core-Systemen ungeeignet, da das De-/Aktivieren der Interrupts **aller** CPU's und das sperren des Systembus extrem teuer ist.

15 Synchronisation (Hardware)

Bei Single-Core-Systemen verwenden wir den Interrupt-based Synchronisation Ansatz.

Auf Multi-Core-Systemen verwenden wir Hardwareanweisungen, wie:

- `test_and_set`
- `compare_and_swap`

und atomare Variablen und Funktionen. Somit haben wir eine Synchronisation durch die Hardware.

- `test_and_set`

```

atomic_bool test_and_set(atomic_bool *lock) {
    atomic_bool old = *lock;
    *lock = true;
    return old
}

```

Beispielnutzung:

```

atomic_bool lock = false;

do {
    // spinlock / busy waiting
    while (test_and_set(&lock));

    // critical section

    lock = false;
    // remainder section

} while (true)

```

- `compare_and_swap`

```

atomic_int compare_and_swap(atomic_int *val,
                           int expected, int new_val) {
    temp = *val;
    if(*val == expected)
        *val = new_val;
    return temp;
}

```

Beispielnutzung:

```

atomic_int lock = 0;

do {
    // spinlock / busy waiting
    while (compare_and_swap(&lock), 0, 1) == 1;

    // critical section

    lock = 0;
    // remainder section

} while (true)

• begrenztes warten mit compare_and_swap

atomic_bool wait[N] = { false };
atomic_int lock = 0;

// Prozess i
do {
    wait[i] = true; // wait[i] = true; Prozess i wartet auf
                    // kritischen Bereich

    // spinlock / busy waiting
    while (wait[i] && compare_and_swap(&lock), 0, 1) == 1;
    wait[i] = false;

    // critical section

    j = (i + 1) % N;
    while((j != i) && !wait[j])
        j = (j + 1) % N;

    if(j == i) lock = 0; // release
    else wait[j] = false; // transfer lock
    // remainder section

} while (true)

```

15.1 Spinlock / Busy waiting

Bei einem Spinlock oder auch busy waiting wartet ein Prozess darauf, dass eine bestimmte Kondition erfüllt wird. Dabei führt er bei jeder Überprüfung einen Rücksprung aus, und prüft erneut. Die Verschwendet CPU und ist schlecht in Timesharing-Systemen. Vorteil ist das kein Contextswitch stattfinden muss. Sie sind außerdem gut für kurzes Warten in Multi-Core-Systemen.

Wie bereits oben gesehen kann durch die Verwendung von Spinlocks eine Synchronisation, auf Kosten von CPU Ressourcen stattfinden.

15.2 Semaphoren

Beschreibung siehe Semaphores 14.16.

15.2.1 Implementierung mittels Spinlock:

```

void wait(sem) {
    while (sem <= 0);
    sem--;
}

```

```

void signal(sem) {
    sem++;
}

```

Beispielnutzung:

```
semaphore sem = 1
```

```

wait(sem)
// critical section
signal(sem)

```

15.2.2 Implementierung ohne Spinlock:

mittels Semaphore Warteschlange:

```

typedef struct _process {
    // true == busy
    bool locked;
    _process *next;
} Process;

typedef struct {
    int value;
    struct Process *list;
} Semaphore;

// last process in the queue
Process *last = NULL;

• lock

lock(Process *l, Process *p) {           // last process
    p->next = NULL;                      // p
    Prozess pred = fetch_and_store(l, p); // p: new tail: pred = l; l = p;
    if(pred != NULL) {                   // Locked; List not empty
        p->locked = true;              // Locked by pred
        pred->next = p;                // insert p
        sleep();                        // wait on pred
    }
}

• unlock

unlock(Process *l, Process *p) {
    if(p->next == NULL) {             // no successor in the list
        if(compare_and_swap(l, p, NULL)) // if (l == p) l = NULL
            return;
        while(p->next == NULL);       // wait in succ
    }
    p->next->locked = false;         // unlock succ
    wakeup(p->next);
}

• process p

void wait(Semaphore *sem) {
    sem->value--;
    if(sem->value < 0)
        lock(last, p);
}

void signal(Semaphore *sem) {
    sem->value++;
    if(sem->value <= 0)
        unlock(last, p);
}

```

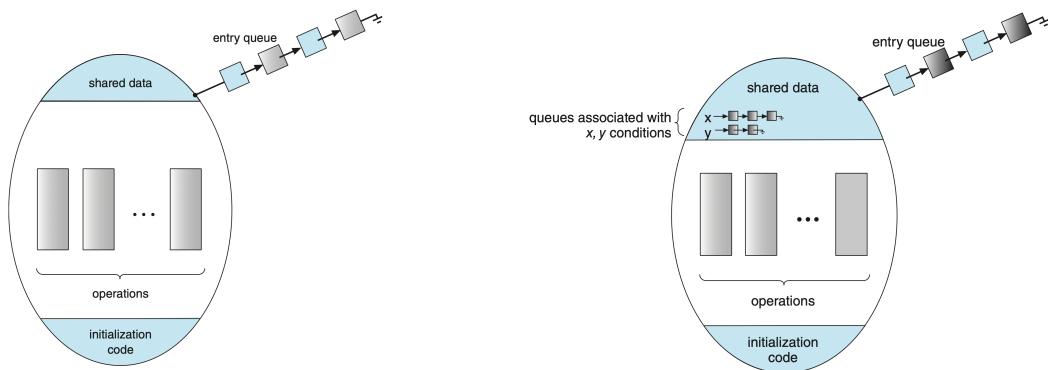
Falls jemand das hier nicht versteht, würde ich gerne auf die Folien von Herr Radush verweisen, mit denen ich genau so viel wie hier steht verstanden habe. (Nicht besonders viel!)

15.3 Monitor und Condition Konstrukt (Java)

Ein Monitor ist ein ADT der Daten und vordefinierte Funktionen bereitstellt. Diese Funktionen sind mutual exclusive. Die Daten auf die Zugegriffen werden kann sind nur lokal vorhanden und somit parallel zugreifbar. Der Monitor stellt sicher das zu jedem Zeitpunkt maximal ein Prozess aktiv die Daten des Monitors manipuliert.

Um dies zu ermöglichen wird ein weiteres Konstrukt, die Condition benötigt. Die einzigen 2 Operationen auf den Conditions sind signal() und wait(), wobei wait mit einem Parameter von Typ Condition aufgerufen werden kann. Dies ermöglicht Bedingtes Warten.

Die Conditions werden beim schlafen bzw. warten auf Ausführung in einer Queue gespeichert.



Beispiel (Serial):

```
monitor Serial {
    condition x = 0;
    bool done = false;
    void f1() {
        Anweisung s1;
        done = true;
        x.signal();
    }

    void f2() {
        if(done == false)
            x.wait();
        Anweisung s2;
    }
}
```

Implementierung mit Semaphoren:

```
monitor ParallelSem {
    semaphore mutex = 1; // Monitor lock
    semaphore next = 0; // Lock of Conditions
    int next_count = 0; // Count Conditions

    Function f() {
        // calculation
        if (next_count > 0)
            signal(next);
        else
            signal(mutex);
    }
}
```

Prozess P_1 :

```
Serial s1;
s1.f1();
```

Prozess P_2 :

```
Serial s2;
s2.f2();
```

```
condition X {
    semaphore x_sem = 0;
    int x_count = 0;

    x.wait() {
        x_count++;
        if(next_count > 0)
            signal(next);
        else
            signal(mutex);

        wait(x_sem);
        x_count--;
    }
}
```

```

        }

        x.signal() {
            if(x_count > 0) {
                next_count++;
                signal(x_sem);
                wait(next);
                next_count--;
            }
        }
    }
}

```

Ressourcenzuteilung:

```

monitor ResourceAllocator {
    boolean busy = false;
    condition x;

    void acquire(int time) {
        if(busy)
            x.wait(time);
        busy = true;
    }

    void release() {
        busy = false;
        x.signal();
    }
}

ResourceAllocator R;
Time t;

R.acquire(t);
// acquire Ressources

R.release();

```

Dining Philosophers Monitor basiert:

```

monitor DiningPhilosophers {
    enum { THINKING, HUNGRY, EATING } state[N];
    condition self[N];

    int left(int i) {
        return (i + N - 1) % N;
    }

    int right(int i) {
        return (i + 1) % N;
    }

    void test_and_eat(int i) {
        if(state[i] == HUNGRY &&
           state[left(i)] != EATING &&
           state[right(i)] != EATING) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    initialization() {
        for(int i = 0; i < N; i++)
            state[i] = THINKING;
    }

    void pickup(int i) {
        state[i] = HUNGRY;
        test_and_eat(i);
        if(state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test_and_eat(left(i));
        test_and_eat(right(i));
    }
}

```

Auch hier habe ich die Folien nicht richtig verstanden und eigentlich nur kopiert!

16 Alternative Ansätze der Synchronisation

Auf neuen Systemen ist wie bereits mehrfach angesprochen ein paralleles Berechnen von Problemen zum Standard geworden. Um dies zu ermöglichen werden meist Mutexes bzw. Semaphoren und andere Konzepte verwendet. Diese sind jedoch nicht sonderlich schnell und schützen nicht vor deadlocks, race conditions und liveness hazards wie zum Beispiel Livelocks.

Außerdem verliert man bei zunehmender Thread Anzahl performance durch erschwerete jedoch nötige Synchronisation.

Um diese Probleme zu lösen wurden neue Wege gesucht um das Programmieren zu vereinfachen und trotzdem die Performance der Programme zu erhalten.

Wir gehen nur auf das Konzept des Transactional Memorys ein. Weitere Ansätze sind OpenMP und Funktionale Programmiersprachen (nachzulesen in der zugrundeliegenden Quelle Chapter 7)

16.1 Transactional Memory

Ein Ansatz ist der Transaktionselle Speicher. Ein Traditioneller Ansatz (mit Mutexes) um Daten im Shared memory zu verändern wäre wie folgt:

```
void update() {
    acquire();
    // update shared memory
    release();
}
```

Da dies jedoch potentielle Deadlocks und andere Fehler mit sich bringt kann auf einen anderen Ansatz gesetzt werden.

Wir erstellen ein neues Konstrukt `atomic{S}`, welches sicherstellt, dass Operationen auf S transaktionell stattfinden.

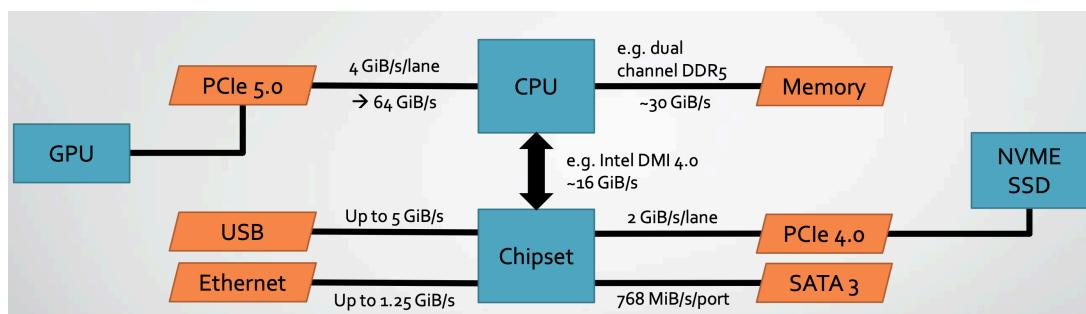
```
void update {
    atomic {
        // update shared memory
    }
}
```

Dadurch ist nichtmehr der Programmierer sonder der Mechanismus an sich zuständig für die Synchronisation der Operation.

Dieses Konstrukt wird meist vom Compiler generiert und durch Hardware Transaktionsspeicher im Cache unterstützt.

17 Input/Output (I/O)

17.1 Architektur



Unter externen Geräten(Devices) verstehen wir alles, was nicht CPU bzw. RAM ist (z.B. GPU, Networkcard, SSDs, ...). Diese Geräte sind meistens bestimmten Speicher Adressen zugeordnet.

17.2 Speicher

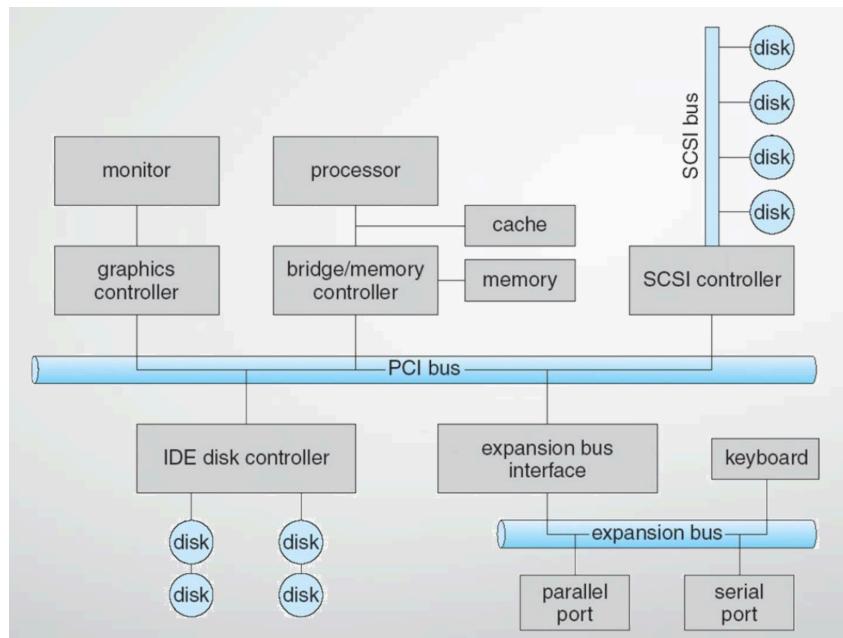
- SRAM (Static Random Access Memory)

Wird meist als Cachespeicher benutzt. Komplex und teuer!

- DRAM (Dynamic Random Access Memory)

Relativ günstig. Dichte Speicher Anordnung. Kapazitoren verlieren Ladung und müssen periodisch neu "beschrieben" werden.

17.3 I/O Bus



Interconnect welches für die Kommunikation mit vielen verschiedenen Geräten gebraucht wird.

17.4 Device Communication

Wir unterscheiden zuerst zwischen Speicher gemapptem I/O (*Memory mapped I/O*), gemappten Gerät Speicher (*Mapped device memory*), Port gemapptem I/O (*Port mapped I/O*) und DMA (*Direct memory access*) unterschieden.

Bei *Memory mapped I/O* wird einem Gerät eine spezifizierte Speicheradresse zugeordnet und read/write kann genau wie auf den restlichen RAM genutzt werden. Die einzelne Semantic ist je nach Gerät spezifisch. Es handelt sich hier jedoch *nicht wirklich um RAM*.

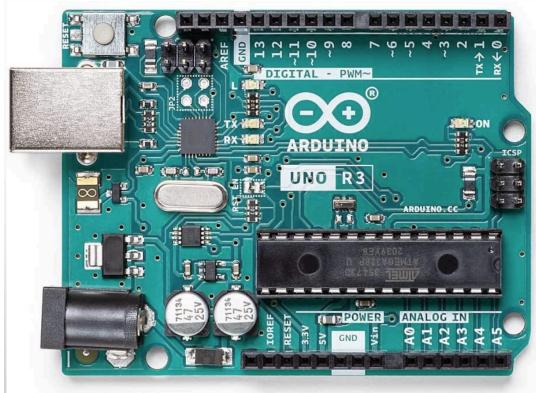
Bei *Mapped device memory* wird eine spezifische Speicheradresse, dem RAM des externen Geräts zugeordnet.

Bei *Port mapped I/O* wird von manchen Architekturen ermöglicht. x86 stellt zu Beispiel die in/out Befehle zur Verfügung mit denen I/O ports/pins gesteuert werden können. Der RAM und die I/O haben einen getrennten Addressraum.

Bei *DMA* schreibt/liest das Gerät automatisch in/aus den/dem RAM. Die CPU ist hier nur der Initiator und kann während dem Transfer andere Aufgaben erledigen.

17.4.1 Beispiel *Memory Mapped I/O* an einem Arduino Uno ATmega328P:

CPU Register, die den Status der I/O pins beinhalten. 8 GPIO (General purpose I/O) pins sind verbunden um einen **GPIO Port** zu bilden.



Der Zugriff erfolgt wie auf normalen Speicher:

```
volatile uint8_t* portb_direction_addr = (volatile uint8_t*)0x24;
volatile uint8_t* portb_addr = (volatile uint8_t*)0x25;

*portb_direction_addr = 0b11111111; // Configure all pins on port B as output

*portb_addr = 0b11111111;           // Set all pins on port B high
*portb_addr = 0b00000000;           // Set all pins on port B low
```

Der ATmega328P hat 3 verschiedene Addressräume.

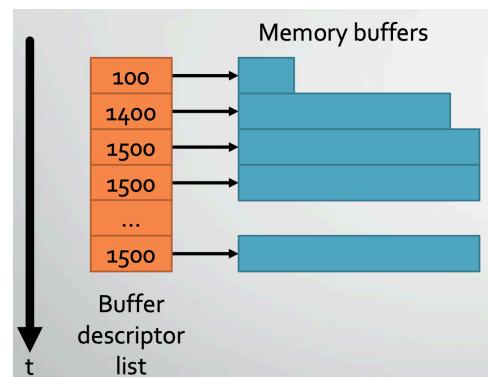
- Regulären RAM: Inklusive dem *Memory mapped I/O*.
- Flash memory: welches das Programm und den Bootloader enthält.
- Non-volatile EEPROM: Dieser kann genutzt werden um permanent Daten zu speichern.

Alle Adressen starten bei 0x0000. Der gewählte Addressraum wird durch eine Assembly Instruktion angegeben.

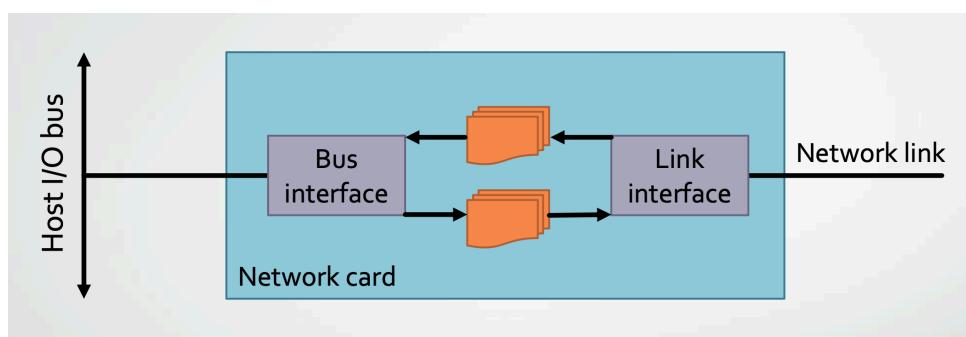
17.4.2 Beispiel DMA-Buffer:

Die CPU wird nur genutzt um den Transfer zu initialisieren und beim Abschluss zu überprüfen. Der Transfer an sich ist unabhängig.

Damit dies möglich ist braucht es gewisse Buffer, damit die initialisierten und in einer Queue wartenden Transfers direkt stattfinden können.



Wir betrachten nun das Beispiel an einem NIC (Network Card Interface).



Das Link-Interface kommuniziert mit der Network hardware. Das Bus-Interface nutzt DMA um Pakete direkt in den RAM zu lesen/schreiben. Für die Operations Queue wird eine FIFO für das Lesen und das Schreiben verwendet. Der Gerätetreiber stellt dem Kernel die benötigte Funktionalität zur Verfügung (reset, ioctl, output, interrupt, read, write, ...). Um das Gerät zu synchronisieren (lesen, schreiben, usw.) kann die einfache Methode des **Polling** oder ein Interrupt-based Ansatz verwendet werden.

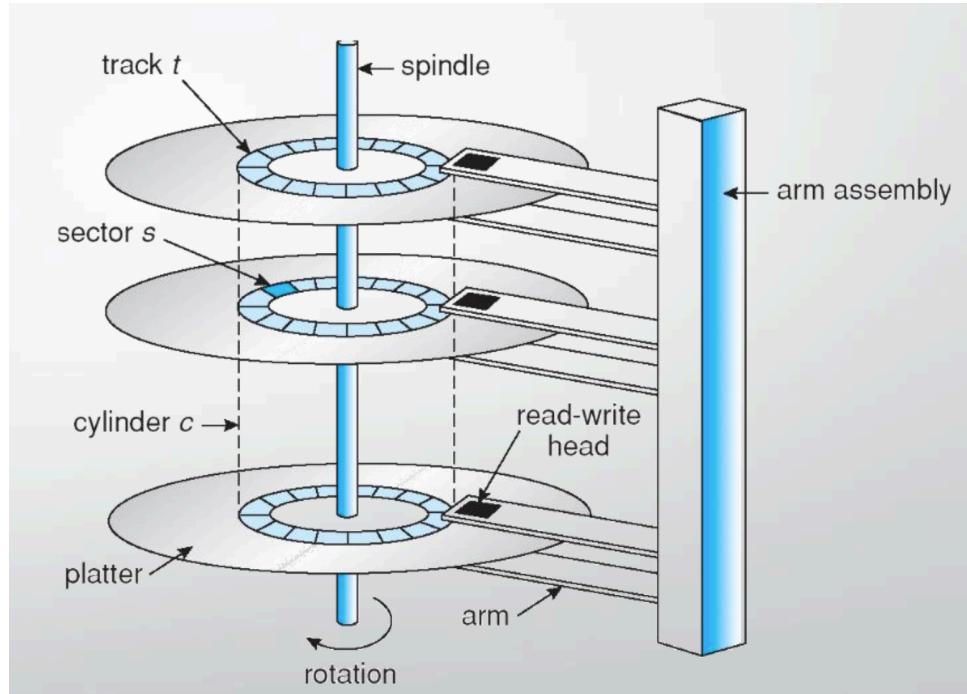
Polling: Überprüft ununterbrochen ob die Operation beendet wurde. Das benötigt viel CPU Resourcen bzw. lässt die CPU busy waiting. Außerdem können keine 2 oder mehr Operation gleichzeitig ausgeführt werden.

Interrupt-based Device Synchronisation: Das Gerät triggert einen CPU Interrupt um etwas zu signalisieren. Der Interrupthandler wird aufgerufen und der Gerätetreiber kann den Grund des Interrupts sofort prüfen. Dadurch können CPU und Gerät unabhängig voneinander Arbeiten. Kommt es zu häufigen Interrupts kann die CPU Geschwindigkeit stark beeinflusst werden, da jedes mal ein Contextswitch getriggert wird. Dieser Ansatz wird von den meisten Betriebssystemen genutzt.

18 Speicher Hardware und Software

18.1 HDD

HDDs (Hard disk drive) sind magnetische Datenträger. Sie nutzen mehrere gestapelte platters, die in konzentrische tracks aufgeteilt werden. Mehrere sectors erzeugen einen track. Die gleichen tracks von allen platters werden cylinder genannt. Die Schreib und Leseköpfe lesen/schreiben auf den cylinder. In einer HDD wird ein großer Teil der Daten für die Error Korrektur reserviert.



Die Geschwindigkeit einer HDD hängt stark von der Ordnung und dem Ort des aufgerufenen Speicher Segments ab. Der sequenzielle Abruf ist wesentlich schneller als der willkürliche Zugriff.

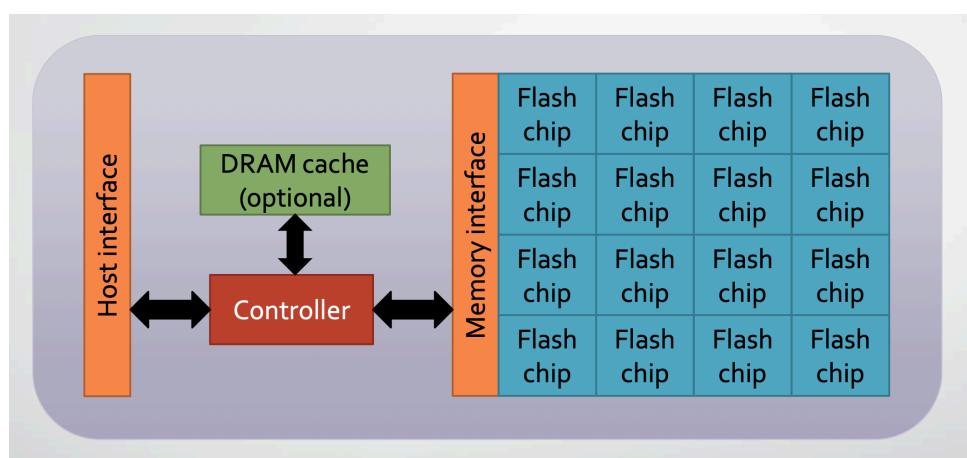
18.2 SSD

SSDs (Solid state drives) sind Flash Datenträger, ohne jegliche Bewegten Teile. Sie haben die HDDs in so gut wie allen Bereichen und Anwendungen ersetzt. Die Daten werden durch elektrische Ladung gespeichert und halten über 10+ Jahre. Speicherzellen können beinahe unendlich oft gelesen werden. Das schreiben ist jedoch auf 10.000 bis 100.000 beschränkt. Eine erhöhte Redundanz und Abnutzungsausgleich kompensiert diesen Nachteil. SSDs sind schnell, zuverlässig und Erschütterungsresistent.

Abnutzungsausgleich: (*Wear leveling*)

Teilt die Schreibvorgänge auf den gesamten Speicher gleichmäßig auf, um die Lebensdauer des Geräts zu verlängern. Dies benötigt eine komplexe Addresslogik, die vom Flash controller erledigt wird. Ist die SSD fast voll kann dies zu Performance Verlust führen.

Architektur:



Der Flash controller ist zuständig den Speicher als Zusammenhängenden Bereich an das System zu übergeben.

18.3 NAND und NOR Flash

NAND Flash ist dicht bedeckt mit einer hohen Kapazität pro Speicherchip. NAND ist anfälliger für Fehler und benötigt Error correction. Die Schreibgeschwindigkeit ist hoch.

NOR Flash hat eine schnelle Lesegeschwindigkeit und kann genutzt werden um Programmcode direkt in-place auszuführen. Es ist zuverlässig und benötigt virtuell keine Error correction. NOR Flash wird häufig für Embedded Systems verwendet.

Typen:

- SLC: (Single) 1 Bit
- MLC: (Multi) 2 Bits
- TLC: (Triple) 3 Bits
- QLC: (Quad) 4 Bits

Das Erhöhen der Bits pro Zelle verringert die Schreibgeschwindigkeit des NAND-Flash. Die Kosten pro Bit sind jedoch geringer.

18.4 RAID

RAID oder auch Redundant Array of Independent Disks ist das kombinieren von mehreren Datenträger zu einem virtuellen Datenträger. Es gibt Unterschiedliche RAID Level, die unterschiedliche Aspekte optimieren. Die Hardware Implementation ist teuer, aber schnell, hat keinen

CPU-Overhead, aber ist nicht portable, da RAID Kontroller von verschiedenen Herstellern verschieden funktionieren. Die Software Implementation ist kostenlos, auf modernen CPUs genau so schnell wie eine Hardware Implementation und portable.

Die Benutzung von RAID ist nicht gleich einem BACKUP!

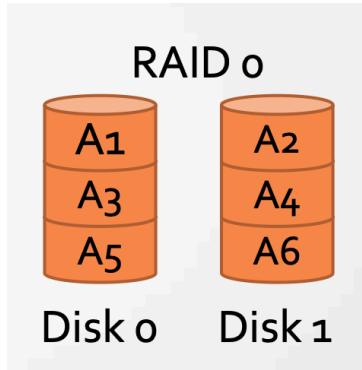
18.5 RAID Standard Levels

$$N = \# \text{disks}$$

$$D = \text{disk size}$$

18.5.1 RAID 0

(*Striping of blocks across disks*)



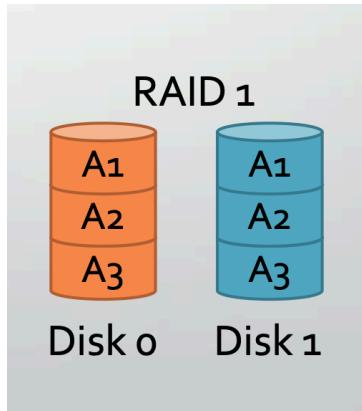
Keine Redundanz. Alle Speicherblöcke werden zu einem großen Block zusammengefasst. Die Lese-/Schreibgeschwindigkeit verbessert sich.

Nutzbare Speicher:

$$N \cdot D$$

18.5.2 RAID 1

(*Mirroring of blocks across disks*)



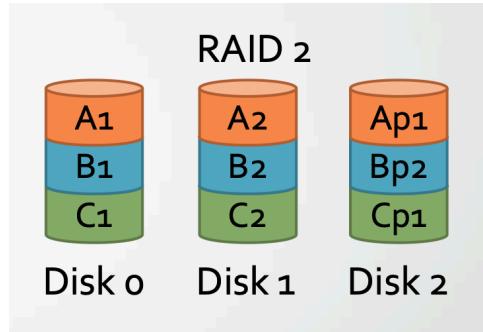
Redundanz bis zum Versagen eines kompletten Datenträgers. Die Lesegeschwindigkeit verbessert sich.

Nutzbare Speicher:

$$D$$

18.5.3 RAID 2

(*Striping of bits with hamming code error correction*)

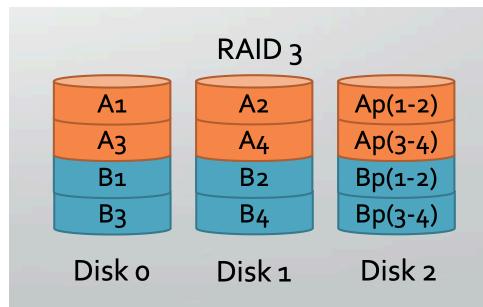


Es werden mindestens 3 Datenträger benötigt. Durch die Error correction ist es möglich, die Daten eines Datenträgers nach dem Versagen wieder herzustellen. Die Lese und Schreib Geschwindigkeit verändert sich nicht. Dieser Ansatz wird in der Praxis so gut wie nie genutzt.
Nutzbarer Speicher:

$$(N - 1) \cdot D$$

18.5.4 RAID 3

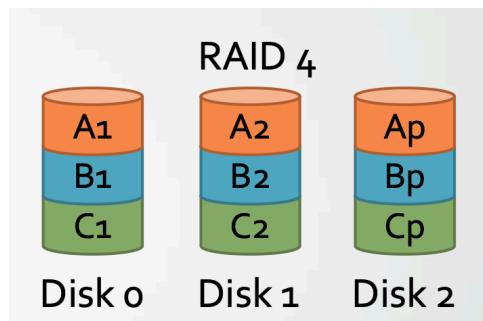
(*Striping of bytes with parity disk*)



Die selben Eigenschaften wie RAID 2, jedoch wesentlich einfachere Berechnung.

18.5.5 RAID 4

(*Striping of blocks with parity disk*)



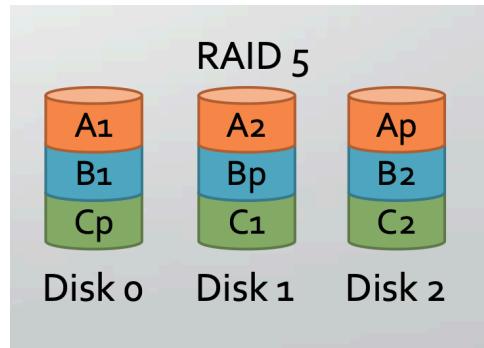
Mindestens 3 Datenträger, komplette Wiederherstellung eines fehlerhaften Datenträgers. Die Schreib Geschwindigkeit ist nicht erhöht, jedoch die Lesegeschwindigkeit. Wird in der Praxis selten genutzt.

Nutzbarer Speicher:

$$(N - 1) \cdot D$$

18.5.6 RAID 5

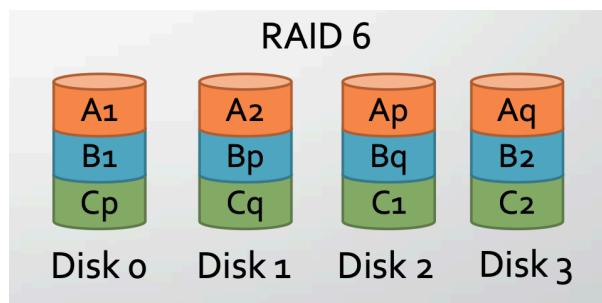
(*Striping of blocks with distributed parity*)



Die selben Eigenschaften wie RAID 4. Durch die Verteilung sind Lese- und Schreibgeschwindigkeit verbessert. Wird häufig genutzt und ersetzt (eigentlich) RAID 2-4.

18.5.7 RAID 6

(*Extension of RAID 5, adding another parity block*)

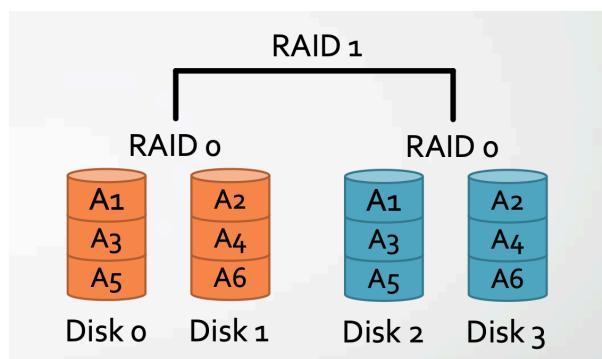


Es werden mindestens 4 Datenträger benötigt. Durch die zweifache Verteilung können 2 fehlerhafte Datenträger komplett wieder hergestellt werden. Wird häufig genutzt um weitere Sicherheit zu gewährleisten. Z.b. nach dem ersetzen eines fehlerhaften Datenträgers.

18.6 RAID Hybrid Levels

18.6.1 RAID 01 / RAID 10

(*Combination of RAID 0 and 1*)



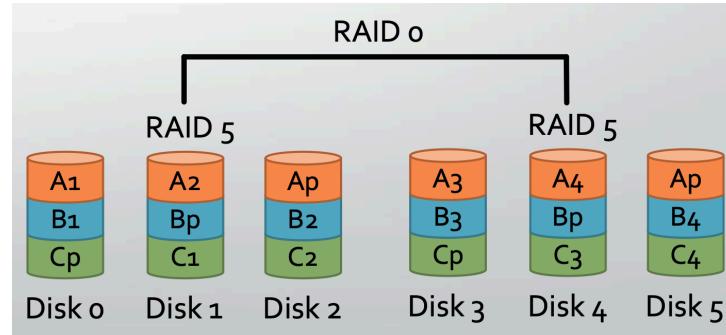
Es werden mindestens 4 Datenträger benötigt. Es handelt sich um eine Kombination von RAID 0 und RAID 1. (*Mirror of stripes*) bzw. (*Stripe of mirrors*).

Nutzbarer Speicher:

$$\left(\frac{N}{2}\right) \cdot D$$

18.6.2 RAID 50

(Combination of RAID 5 and 0)



Es werden mindestens 6 Datenträger benötigt. In jedem RAID 5 Block kann ein Datenträger versagen und komplett wieder hergestellt werden. Die Schreibgeschwindigkeit wird erhöht.

Nutzbarer Speicher:

$$M = \#\text{RAID groups}$$

$$M \cdot \left(\frac{N}{M} - 1\right) \cdot D$$

19 Filesystems und Partitioning

Die Datenträger eines Geräts werden dem Betriebssystem als lineares Array an bytes übergeben. Um die einzelnen Daten des Systems zu speichern, wieder zu finden, und viele weitere Operationen auf ihnen auszuführen, wird ein Dateisystem genutzt.

19.1 Partitiontable

Die Partitionstabelle befindet sich am Anfang des Datenträgers bzw. des Speicherarrays. Sie enthält die Position, den Typ, den Name, usw. von jeder Partition.

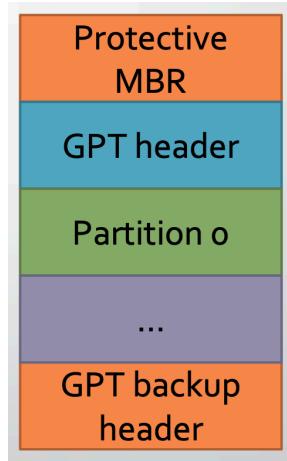
19.1.1 MBR

MBR (Master Boot Record) ist eine simple Partitionstabelle. Sie startet im boot sector (ersten 512bytes) und kann die Partitionsdaten von Datenträgern bis zu 2TB speichern. Bei einer MBR Partitionstabelle ist die Anzahl der Partitionen zudem auf 4 Partitionen beschränkt. MBR wird heutzutage nicht mehr genutzt. Der heutige Standard ist **GPT**.

19.1.2 GPT

GPT (GUID Partition Table) ist eine weitere Art der Partitionstabelle. Sie wird heutzutage hauptsächlich genutzt.

Im Bootsector wird zuerst ein Bereich von 512bytes freigehalten. Dies ist dazu da um Programme daran zu hindern die GPT Tabelle durch schreiben von MBR Daten zu zerstören. Der GPT-Header beinhaltet Informationen zu den einzelnen Partitionen. Mit einer GPT Tabelle sind virtuell unendlich viele Partitionen möglich. Jeder Partition wird durch eine GUID gekennzeichnet. Am Ende des Datenträger findet sich zudem eine Backup Header, der bei Datenkorruption am Anfang des Datenträgers potenziell die Partitionen wieder herstellen kann.



19.2 FAT Filesystem

FAT (File Allocation Table) ist ein simples aber dementsprechend naives Dateisystem. Der Datenträger ist in gleich große Cluster unterteilt (meist 16KiB). Eine Datei wird durch eine linked-List von einem oder mehr dieser Cluster dargestellt. Eine Indexierungstabelle zu Beginn der Partition zeigt auf das Startcluster einer Datei. Folders sind spezielle Dateien die auf die Dateien/cluster in dem Folder zeigen. Das FAT Dateisystem schützt nicht gegen Datenkorruption und hat keinerlei error detection bzw. data recovery.

FAT wird heutzutage noch in manchen Embedded Systems verwendet.

19.3 Journaling Filesystem

Journaling Dateisysteme wie NTFS oder ext4 werden heutzutage häufig verwendet. Sie schützen vor datacorruption. Das schreiben auf das Dateisystem ist jedoch nicht besonders schnell und benötigt mehrere Schritte. Fehler während des Schreibens sind somit gefährlich und können Daten bzw. im schlechtesten Fall Dateisystem Metadaten zerstören.

Wie das Wort Journaling schon vermuten lässt werden Änderungen an Dateien in einem *Journal* gespeichert. Dies passiert als erstes bei einer Schreiboperation. Wenn nun ein Fehler aufgetreten ist können die Daten durch das Journal wieder ersetzt werden.

Fehler im Journal werden durch das inkludieren der checksum in den Journal Einträgen verhindert.

19.4 Multi-Disk Filesystem

Multi-Disk-Filesystems kombinieren Ideen von RAID und dem Journaling Dateisystem. Beispiel für diese Dateisysteme sind ZFS oder btrfs. Das Dateisystem kann hier über mehrere Datenträger verteilt sein. Dies passiert ohne ein RAID Array. Diese Dateisysteme schützen vor bit-rot durch das Speichern der checksum von jeder Datei. Duplikate werden nur einmal auf dem Dateisystem gespeichert und durch Referenzen zugänglich gemacht.

20 Scheduling