Weiter zu Kapitel 3 + Kapitel 4 -Langlebigkeit und Dateisysteme (Persistence) :

Problem: mit Korrektheit in Lock-Implementierung -> **Wakeup/Waiting Race**:
Lsg: mit **setpark()**, mit der ein Thread anzeigen kann, dass er gleich schlafen geht|-wird Thread nach Aufruf setpark() unterbrochen (Interrupt) und anderer Thread führt unpark() aus bevor park() aufgerufen, dann wird park() ignoriert
->Mit Queue wird Reihenfolge festgelegt. (Ist mit setpark() auch fair). Performance auch besser + park() und unpark()

```
typedef struct lock_t { int flag; int guard; queue_t *q; } lock_t;
void lock_init(lock_t *m) { m->flag = 0; m->guard = 0; queue_init(m->q); }
void lock(lock_t *m) {
while (TestAndSet(&m->guard, 1) == 1) ; // Nicht-atomares lock() synchronisieren
if (m->flag == 0) { // Lock frei -> wird beansprucht
m->flag = 1;
m->guard = 0; } else {
queue_add(m->q, gettid()); // Lock beansprucht -> In Warteschlange
setpark(); // Wakeup Race verhindern!
m->guard = 0; // guard macht Spin-Lock für nicht-atomares lock()
park(); // Thread geht schlafen
} }
void unlock(lock_t *m) {
while (TestAndSet(&m->guard, 1) == 1) ; // wie lock()
if (queue_empty(m->q)) m->flag = 0; // queue leer -> lock wird freigegeben
else unpark(queue_remove(m->q)); // nächster Thread aus Queue (wird geweckt)
m->guard = 0; } // lock wieder frei
```

Aus der Praxis: Futex Locks in Linux: -Jedem Fast Userspace MuTEX (Futex) ist spezifische PAS sowie (Kernel-interne) Queue zugeordnet |-Als atomare Operation wird Compare-And-Block verwend. -Funktion **futex_wait** (address, expected) versetzt den aufrufenden Thread in Ruhezustand, wenn der Wert an address,Wert von expected entspricht |-Funktion **futex_wake** (address) weckt wartenden Thread aus Warteschlange auf |-Es wird 32 Bit Ganzzahl verwendet, in der erste Bit den Zustand des Locks kodiert und die verbleibenden Stellen die Anzahl der wartenden Threads

Two-Phase Lock: Spinning und Yielding: Erkenntnis: Spinning kann nützlich sein (zB: wenn Lock kurz vor Freigabe steht)
Phase1: Thread macht eine Weile Spin-Wait, damit er den Lock schnell erhält ->Wenn Lock nicht erhalten wird, wird -> dann Phase2: Thread wird schlafen gelegt und erst geweckt, wenn der Lock wieder freigegeben wird

---

Zwstand: kennen nun Möglichkeiten um Sperren auf kritische Abschnitte zu setzen -> Naiver Ansatz: Spin-Wait -> anders:
Bedingungsvariablen: Man will mit Thread auf Eintreten einer Bedingung warten. Spin-Waits teilweise inkorrekt und v.a. ineffizient -> Bedingungsvariable ist explizite Warteschlange mit: **wait()** legt Thread in Warteschlange, wenn Bedingung nicht erfüllt ist | **signal()** weckt einen oder mehrere Threads auf, wenn sich Zustand ändert.
BspImplementation: -> ***done*** benötigt, um dauerhaftes Schlafen zu verhindern, wenn ***signal()*** vor ***wait()*** aufgerufen wird
->Man braucht Lock vor ***wait()/signal()***, um Race Conditions bzgl. done zu verhindern. Sonst Parent für immer schlafen
->Man muss ständig while() (statt if()) machen wegen Producer/Consumer Problem (auch: Bounded Buffer Problem)
Producer/Consumer Problem: - Erzeuger (Producer) produziert Datenelemente und legt diese in begrenzten Puffer ab - Verbraucher (Consumer) entnimmt Elemente aus Puffer und verarbeitet sie ->der Puffer wird von Erzeuger und Verbraucher gemeinsam genutzt und muss synchronisiert werden
->Es kann vorkommen, dass sich ein anderer Thread vordrängelt und die Daten manipuliert, dann muss erneut geprüft werden, ob sie den gewünschten Status haben.
Zusatz: Außerdem kann es bei der zweiten Implementation (Eine Bedingungsvariable, ein Lock und while-Schleifen) vorkommen, dass sich alle Schlafen legen. Dritte Implementation ist korrekt (Mit zwei Bedingungsvariablen). Änderungen werden noch gemacht, um mehr im Puffer zu speichern.
-**Producer/Consumer mit Synchronisierung:** Für synchronisierten Zugriff bauen wir eine Bedingungsvariable, Lock und if-Bedingungen ein ->Variante funktioniert für je einen Producer- und Consumer-Thread (nicht für 2 Consumer)
->Vers1: 2 Consumer (Tc1 , Tc2) und ein Producer (Tp): -Nachdem Tp geweckt wird besteht Möglichkeit, dass sich Tc2 vordrängelt und Puffer vor Tc1 konsumiert, da Tc2 nie "schlafen" gelegt wurde |-Tc1 wurde zwar "geweckt" (Sleep) aber der Scheduler muss ihn erst mal ausführen (Run).
->Vers2: --- mit while-Schleifen: Tc1 weckt anstelle des Producers Tp einen weiteren Consumer Tc2. Da der Puffer bereits leer ist, legt sich Tc2 ohne weitere Aktion wieder schlafen (while). ->Lsg:
->Vers3: Für den synchronisierten Zugriff bauen wir zwei Bedingungsvariablen, einen Lock und while-Schleifen ein -> korrekte Producer-/Consumer-Lösung gefunden (noch nötig: Nebenläufigkeit und Effizienz bei vielen Threads) ->Lsg: brauchen mehr Platz im Puffer -> Consumer und Producer auf veränderte Puffer-Größe anpassen .....

---

weitere Synchronisierungsprimitive: Locks und Bedingungsvariablen nötig, um einer Reihe von Problemen der Nebenläufigkeit aus dem Weg zu gehen
Semaphor (Semaphore): ist Objekt mit Integer-Wert, den man mit Methoden ***sem_wait() + sem_post()*** manipulie können

Pseudocode Semaphore:

```
int sem_wait(sem_t *s) { *s = *s -1; // decrement s (atomar)
// Wenn: s negativ => Warten (anzahl an waiting threads)
} int sem_post(sem_t *s) { *s = *s +1; // increment s (atomar)
// Wenn: Min. 1 Thread wartet => 1 Thread aufwachen
}
```

Bemerk: -**sem_wait()** wird entweder sofort zur Rückgabe gehen (wenn s Wert 1 oder größer hat) oder rufende Instanz in Warte-Zustand versetzen |-Wiederholte Aufrufe verschiednen Threads landen entsprechend in einer Queue|-**sem_post()** wartet nicht Erfüllung einer bestimmten Bedingung -> wird lediglich Wert von s erhöht und dann wartender Thread geweckt |-Ist Wert der Variable s negativ, dann entspricht der Betrag des Wertes der Anzahl wartender Threads
Bsp1: Semaphore als Lock: es ist korrekter Lock; T1 geht schlafen wenn er versucht blockierten Lock zu beanspruchen
Bsp2: **Semaphore als Bedingungsvariable**: -X sollte Wert 0 annehmen -> Methode sem_wait() blockiert nur wenn s < 0 -Andernfalls würde der Parent-Thread nicht warten
->Fallunterscheidung: **Fall 1**: Parent ruft sem_wait() auf, bevor Child sem_post() aufgerufen hat
Fall 2: Child läuft bis zum Ende durch, bevor Parent sem_wait() aufgerufen hat
->Semaphoren bezüglich des Producer/Consumer-Problems: Methoden put() und get() mit einem Puffer für mehrere Daten, wie wir sie am Ende des vorigen Abschnitts zu Bedingungsvariablen kennengelernt haben
Bsp3: Semaphore als Bedingungsvariable im Producer/Consumer-Kontext: MAX=1, je ein Producer/Consumer - Ja
MAX=1, mehrere Producer/Consumer - Ja
Problem: MAX>1 und mehreren Consumern/Producern gibt es Race Condition bezüglich unseres Puffers -> denn put() und get() sind nicht synchronisiert ->Werte änderbar von weiteren Producer bevor sie von Consumer gelesen wurden
Lsg: **Mutual Exclusion**: brauchen einen Lock für die Methoden put() und get(), da diese kritische Abschnitte darstellen
Problem: Wenn beide Methoden mit Lock versehen, mögl dass alle Threads aufeinander warten -> **Deadlock** entsteht (P wartet auf mutex gehalten von C. Das wartet auf full gehalten von P).
Fine-Grained Locking: put() und get() weiterhin kritisch und müssen mittels Lock geschützt werden -> Deswegen müssen Locks präzise abgestimmt sein ->In Praxis: würde wahrscheinlich sogar erst innerhalb von put() und get()synchronisiert, um nur dann zu sperren, wenn es nötig ist
Ergebnis: korrekt funktionierende Synchronisierung mittels Semaphoren für beliebige Anzahl an Producern + Consumern
Initialer Wert einer Semaphor: Semaphor kann mit verschiedenen Werten initialisiert werden -> Mit initialen Wert 1 fungiert sie als Lock, mit einem initialen Wert 0 gibt sie eine Reihenfolge vor
->bei Lock stellen wir Ressource (die Sperre) direkt am Anfang zur Verfügung, damit nutzbar |->bei Producer/Consumer gibt es erst keine nutzbare Ressource (Wert: 0) -> erst wenn ein Child-Thread fertig ist, wird eine Ressource zur Verfügung gestellt (Inkrementieren um 1)
Philosophen-Problem: Denken und Essen mit begrenzten Gabeln: Synchronisierungsproblem (Streit um Gabeln). Man will kein Deadlock und kein Aushungern. ->Idee: Gabeln sind dann Ressource, auf die man wartet, wenn man sie noch nicht hat, wenn man Gabel hatte, dann Später zu Verfügung stellen -> immernoch Problem -> Dijkstra's Lösung
Überblick Semaphoren: -Semaphoren sind leistungsfähiges und flexibles Werkzeug zum Schreiben nebenläufiger Programme |-werden aufgrund ihrer Einfachheit und Nützlichkeit oft anstelle von Locks und Bedingungsvariablen eingesetzt |-Bedingungsvariablen lassen sich jedoch deutlich schwerer damit implementieren als Locks

---

Kapitel 4: Persistence

Persistenz: Man braucht Geräte wie Festplatten, um Daten für lange Zeiträume zu speichern (und bei Stromausfall)
Elementare Fragen: • Wie sollte I/O in Systeme integriert werden? • Was sind entscheidenden Mechanismen dahinter? • Wie können wir diese effizient gestalten?
Situation im Rechner: Memory Bus (CPU mit MEM, am schnellsten), General IO Bus (Netzwerk, NVMe, Grafik, schnell), Parallel IO Bus (Eingabegeräte, SCSI, SATA, USB, langsam). Wird aufgrund von Platz + Kosten so gemacht.
kanonisches Gerät: betrachte: Ein-/Ausgabegeräte auf konzeptueller Ebene -> Geräte brauchen Schnittstelle
Hardware-Schnittstelle: •***Statusregister***: aktuellen Status des Geräts auslesen •***Befehlsregister***: Gerät anweisen, eine bestimmte Aufgabe auszuführen •***Datenregister***: Übertragung von Daten vom/zum Gerät
->Schnittstelle erlaubt es der Systemsoftware Betrieb von Ein-/Ausgabegeräten zu steuern (ähnlich zu Software-APIs)
Hardware-Aufbau: •***Einfache Geräte***: verfügen über einen oder wenige Hardware-Chips zur Implementierung ihrer Funktionalität •**Komplexe Geräte:** hat min. einen Prozessor mit komplexer Firmware, Speicher und weitere Chips
Kanonisches Protokoll: Einheitliche Schnittstelle:

```
while (STATUS == BUSY) ; // Warten solang Gerät Busy ist (Polling!)
data > DATA REGISTER // Daten in Reg schreiben
command > COMMAND REGISTER // Command in Reg schreiben
(starts the device and executes the command) // Command ausführen
while (STATUS == BUSY) ; // Auf Ausführung warten (Polling)
```

Vorteile: -einfach und verständlich -Korrektheit gegeben
Nachteil: -Wiederholtes Lesen des Status (Polling) |-Hauptprozessor ist zu viel an Datenübertragung beteiligt
Jobwechsel: Statt Polling kann Interrupt eingesetzt werden. Aber nur sinnvoll bei langsamen Geräten, da Kosten für Context-Switch hoch sind. Wenn Gerät fertig ist, wird HW-Interrupt ausgelöst. OS handelt das
Strategien für praktischen Einsatz: **1**-Wiederholtes Polling bei Netzwerkgeräten vermeiden -> hoher Paketanzahl kann Interrupt-Livelock auslösen (nur Interrupts werden behandelt) ->Gelegentliches Polling für bessere Kontrolle der Ver-|**3**-Hybrider (zweistufiger Ansatz) für unbekannte Geräte: Erst Polling, wenn es lang dauert, Interrupts.
-> Durch **Programmed I/O (PIO)** verbringt Prozessor zu viel Zeit damit, Daten von und zu Geräten zu verschieben.
-> Taktzyklen des Prozessors zu wertvoll dafür -> müssen die Arbeit auslagern!
Speicherdirektzugriff (Direct Memory Access, DMA): Eine DMA-Engine ist spezielles Gerät, das den Informationsfluss zwischen Hauptspeicher und Ein-/Ausgabegeräten ohne große Beteiligung des Prozessors verwaltet.
Ablauf mit DMA-Engine: 1-Betriebssystem konfiguriert DMA-Engine bezüglich Quell-/Zieladressen in Hauptspeicher und Gerät, sowie Datenmenge bzw. -länge |**2**-Ist Speicherdirektzugriff abgeschlossen, wird passender Interrupt ausgelöst, um Betriebssystem zu informieren

Kommunikation: Mit Interrupts und DMA mögl Schnittstellen von Geräten Bedienen ->Kommunikation mit Gerät:
Ein-/Ausgabebefehle (I/O Instructions): •gibt explizite Befehle für Ein- und Ausgabe (zB: in und out bei x86) |•IdR geht es um Befehle mit priviligierten Rechten |•Betriebssystem kann als einziges mit den Geräten direkt kommunizieren
Memory-Mapped I/O: •Geräte-Register werden auf Adressbereich des Hauptspeichers abgebildet |•Werden durch Lesen und Beschreiben der Speicheradressen angesprochen |•Typische Schnittstellen sind hier etwa load und store ->hat Vorteil: es braucht keine zusätzlichen Befehle zur Unterstützung ->beide Ansätze Anwendung in Praxis
Spezifikationen abstrahieren: Betriebssystem sollte geräteneutral gestaltet werden. Details abstrahiert + ausgelagert:
Gerätetreiber: Alle spezifischen Geräteinteraktionen sind im Gerätetreiber gekapselt. Nur ein im Betriebssystem installierter Gerätetreiber weiß im Detail, wie ein Gerät genau funktioniert
->Anwendungen nutzen meist POSIX-Schnittstelle (Portable Operating System Interface) des Dateisystems. Spezielle Anwendungen (CheckDisk, Defrag) können mittels RAW-Schnittstelle direkt auf Blöcke zugreifen; Auch Gerät mit speziellen Funktionen muss generische Schnittstelle zum Rest des Kernels haben, wobei Funktionen ungenutzt bleiben

---

Dateien und Ordnerstrukturen: Dateisysteme weitere Virtualisierung von physischen Ressourcen. Grundlagen bilden Abstrahierungen Datei und Ordner
Datei: Lineares Array aus Bytes. Können gelesen+geschrieben werden. Jede Datei hat (Low-Level)-Namen: Inode-Nummer
Ordner: Hat Liste aus Werte-Paaren mit (High-Level) Namen und Inodes. Es lassen sich beliebige Verzeichnisbäume aufbauen -> Ordner hat auch Inode-Nummer
Interaktion mit Dateien: Schnittstelle (Datei erstellen (in C)): **1**.Parameter: Dateiname (High-Level) |**2**.Param: Optionen: ***O_CREAT*** ->Erstelle Datei, wenn nicht vorhanden; ***O_WRONLY*** ->Öffne nur zum Schreiben; ***O_TRUNC*** ->Wenn vorhanden, lösche den Inhalt |**3**.Param: Berechtigungen: ***S_IRUSR|S_IWUSR***: Eigentümer darf lesen und schreiben. Rückgabe: File Descriptor: Integer Wert (innerhalb des Prozesses) -> Innerhalb des Prozesses kann man auf offene Dateien zugreifen
-> int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR); -> mit strace wird Systembefehl angezeigt
-> Jeder Prozess hat immer drei „Dateien" offen (**stdin, stdout, stderr**) -> Man kann mit lseek(int file, off_t offset, int whence) nur Teile einer Datei lesen -> Zu gesuchten Teilen kommt man mit Offset ->Dateien speichern/löschen/create:
Systemaufrufe write(): -garantiert nicht, dass Speichern sofort passiert |-System wird angewiesen, Datei demnächst in persistenten Speicher zu schreiben ->Aus Performancegründen üblicherweise im RAM gepuffert und gebündelt geschrieben |->Stürzt System ab, bevor Daten aus dem Puffer tatsächlich geschrieben wurden, dann gehen sie verloren
Systemaufruf fsync(): System wird angewiesen, alle noch nicht geschriebenen Daten aus RAM Puffer in persistenten Speicher zu schreiben. Aufruf terminiert, wenn alle Schreibvorgänge abgeschlossen sind.
Systemaufruf mkdir(): Wird Ordner neu erstellt, ist er leer. Hat allerdings zwei Einträge: Referenz auf sich selbst (.) und Referenz auf übergeordneten Ordner (..)
Systemaufruf rmdir(): Muss leer sein, damit er löschbar. Darf nur Referenzen auf Sich und übergeordneten Ordner haben.
Systemaufruf link(): Neuer Dateiname wird mit bereits existierenden verlinkt. Damit gibt es weiteren Namen (bzw. Pfad) für selbe Datei, da beide Dateien auf selbe Inode zeigen (Datei wird also nicht kopiert)
Systemaufruf unlink(): Referenz zwischen Dateinamen und einer Inode wird entfernt. Aufruf prüft Anzahl der Referenzen auf Inode. Ist Anzahl 0, wird Inode und zugehöriger Speicherbereich freigegeben. (rm verwendet unlink)
->Bei **link()** wird **Hard Link** verwendet, der nur für Dateien unterstützt wird, da bei Ordnern Zyklen auftreten können.
->Außerd können Hard Links nicht in andere Dateisysteme linken, da Inodes nur innerha eines Dateisystem eindeutig sind
Symbolic (Soft) Links: Spezieller Dateityp, der Verweis auf anderen Dateityp enthält. Ziellose u. zyklische Verweise erlaubt.

Einfaches Dateisystem (vsfs): Was muss konkret beim Bauen von Dateisystems beachtet werden? Very Simple File System ist vereinfachte Form von UNIX-Dateisystems.
Datenstrukturen: Welche Datenstrukturen werden vom Dateisystem von der Hardware benötigt (on disk), um Daten und Metadaten zu organisieren? ->auf Arrays von Blöcken beschränken; fortgeschrittene Systeme nutzen Baumstruktur
Zugriffsmethoden: -Wie werden Aufrufe eines Prozess (open(), read(), write()) auf die Datenstrukturen abgebildet? |-Welche Datenstrukturen müssen bei Operation gelesen/geschrieben werden? |-Wie effizient umgesetztbar?
Grundlegender Aufbau: Einteilen des Speicherplatz in Blöcke je 4KB. Meisten Blöcke enthalten Daten, aber am Anfang müssen Metadaten gespeichert werden: -***Inode-Tabelle (I)***: Inode (256B) verwaltet Zeiger auf Datenblöcke/-größe, Eigentümer- /Zugriffsrechte, Zugriffs-/Änderungszeiten. Jeder 4KB-Block kann 16 Inodes verwalten |-***Bitmaps (d) und (i)***: Zuweisungsstrukturen für Daten (d) und Inodes (i) |-***Superblock (S)***: Informationen über Dateisystem: Anzahl an Inodes und Datenblöcken im System (frei, belegt, max), Startadressen der Inode-Tablle und Bitmaps, und Magic Number zur Identifikation |-***Datenregion (D)***: Nutzerdaten (56 Blöcke)
Inhaltsverzeichnis: Elementar für schnelles Finden von Daten sind Inodes (Index Nodes). Jede Inode ist durch Nummern referenziert (Inode Number), mit denen direkt zur richtigen Position gesprungen werden kann.
Beispiel: Inode Nummer 32 -> **1)** Versatz berechnen: 32* sizeof(inode) = 8192B |**2)**Startadresse der Inode-Tabelle addieren (inodeStartAddr = 12KB) = 20KB |**3)** Sektoradresse (Laufwerke mittels Sektoren je 512B adressierbar) berechnen: 20KB/512B -> Sektor 40 (0-indexed) --> Metadaten in Inodes: ***Typ*** (reguläre Datei, Ordner, Symlink, ...); **Größe** (Anzahl belegter Blöcke), ***Berechtigung***, ***Zeit*** (Create, Update, ...), ***Datenblockposit*** (z.B mit Zeiger; aber ist Designentscheidung)
->Multi-Level Index: Indirekte Zeiger werden verwendet. Statt auf einzelnen Block mit Daten, wird auf Block mit weiteren Zeigern verwiesen, die dann die Daten referenzieren (Dadurch wird max. Dateigröße viel höher). Die meisten Dateien sind aber sehr klein, also viel Direkte Blöcke, damit es schneller läuft.
Extend Mapping: Verwendung von Zeigern auf Blockanfang. Zusätzlich wird Länge des Blocks benötigt, um Ausdehnung (Extend) des Blocks zu kennen. z.B. in ext4 und XFS.
Bemerk: -Weniger flexibel, aber kompakter als Multi-Level Index, da weniger Metadaten benötigt werden. -Funktioniert nur gut, solang ausreichend freier zusammenhängender Speicher vorhanden ist. (Dann können Daten zusammenhängend angeordnet werden, ansonsten kommt es zu Fragmentierung)
Linked Allocation: Jede Datei ist eine verkettete Liste aus Blöcken. Das Ende eines Blocks zeigt auf den Anfang des nächsten. Der Ansatz wird im Dateisystem FAT (File Allocation Table) verwendet
Bemerk: -Jeder Ordnereintrag hat Zeiger ersten Block einer Datei |-Keine externe Fragmentierung, da jeder freie Block genutzt werden kann |-Angabe der Dateigröße unnötig und Dateien können einfach wachsen. Aber Random Access ist sehr ineffizient.
VSFS und Ordner: Prinzipiell sind die eine Liste aus Paaren mit Dateinamen und Inode-Nummer -> aber noch mehr Info: -Inode Nummer |-Dateiname |-Länge der Zeichenkette (Dateiname) |-Dateilänge.
-> Aus Sicht des Dateisystems sind Ordner nur spezielle Dateien mit Typ „Directory" ->Ordner hält Datenblöcke, welche von der Inode referenziert werden -> Einfache Liste mit Ordnereinträgen ineffizient. XFS nutzt z.B. B-Bäume.
->Aufbau eines Dateisystems mit Dateien und Ordnern: siehe Folien; Bild
Bemerk: -Virtuelle Speicherverwaltung essentiell für FS. Zu jeder Zeit muss klar sein, welche Inode und Datenblöcke frei sind. Dazu können Bitmaps, Free Lists, (balancierte) Bäume, usw. genutzt werden.
-Beim Bitmap-Ansatz in vsfs wird nach einer freien Position gesucht (Bit nicht gesetzt). Wird Speicher für neue Datei alloziert, wird entsprechender Eintrag in Bitmap auf 1 gesetzt. Entsprechende Aktionen müssen zur Allokation von Datenblöcken ausgeführt werden.
-Um Performance zu verbessern, werden beim Erstellen einer Datei mehrere zusammenhängende Blöcke alloziert. Pre-Allocation reduziert auch die Fragmentierung.
Abläufe: **open(): Finde Inode von bar: 1**-Lesen von Wurzel-Inode (bekannt) |**2**-Lesen des Datenblocks von /, um Inode von /foo zu finden |**3**-Lesen der Inode von /foo |**4**-Lesen des Datenblocks von /foo, um Inode von /foo/bar zu finden. | **5**-Lesen der Inode von /foo/bar.
->Zum Schluss kann Inode von bar in dynamischen Speicher geladen werden, wo finale Berechtigungsprüfung stattfindet.
read(): Lese Inhalte von bar: 1-Lesen der Inode um Adresse des Zielblocks zu finden |**2**-Aktualisiere Zugriffszeit in Inode | **3**-Aktualisiere Eintrag in Tabelle der offenen Dateien für File Deskriptor (vorallem: Versatz/Offset aktualisieren) Nach Abschluss des Lesevorgangs wird close() aufgerufen und File Deskriptor wird wieder freigegeben.
open(): Finde Inode von bar (zum Schreiben): **1**-Lesen und Schreiben der Data Bitmap, um neu allozierte Blöcke als belegt zu markieren |**2**-Lesen und Schreiben der Inode, um neue zugehörige Blocke zu vermerken |**3**-Schreiben der Datenblöcke
Erstellen von Inode: Zuerst Inode und genug freien Speicherplatz im Ordner, der Datei beinhalten soll, allozieren. Dann: **1**-Lese Inode Bitmap (finde freie Inode) |**2**-Schreibe Inode Bitmap (Belegung) |**3**-Schreibe in Inode (Init.) |**4**-Schreibe in Data Bitmap des Ordners (Dateiname, Inode) |**5**-Lesen und Schreiben der Ordner Inode
->Wenn Datenbereich des Ordners wachsen muss, werden weitere IOs benötigt.
Zwischenspeichern: Caching im RAM Mittel, um Kosten der Lesevorgänge zu reduzieren -> Erstes Öffnen verursacht weiterhin viele I/Os (Ordner-Inodes und –Daten lesen), aber danach kann der Cache genutzt werden -> 2 Ansätze:
Static Partitioning: Es wird im Arbeitsspeicher ein Cache mit fester Größe definiert (z.B. 10%). Dieser Speicherbereich kann nicht mehr für andere Aufgaben verwendet werden
Dynamic Partitioning: Die Pages der virtuellen Speicherverwaltung und Pages des Dateisystems bilden gemeinsam einen sogenannten ***Unified Page Cache***.
->Dynamischer Ansatz flexibler, aber komplizierter in Umsetzung ->Außerdem Konkurrenzkampf um Speicherressourcen

```
(starts the device and executes the command) // Command ausführen
while (STATUS == BUSY) ; // Auf Ausführung warten (Polling)
```

(Polling) |-Hauptprozessor ist zu viel an Datenübertragung beteiligt

Jobwechsel: Statt Polling kann Interrupt eingesetzt werden. Aber nur sinnvoll bei langsamen Geräten, da Kosten für Context-Switch hoch sind. Wenn Gerät fertig ist, wird HW-Interrupt ausgelöst. OS handelt das

Strategien für praktischen Einsatz: **1-**Wiederholtes Polling bei Netzwerkgeräten vermeiden -> hoher Paketanzahl kann Interrupt Livelock auslösen (nur Interrupts werden behandelt) ->Gelegentliches Polling für bessere Kontrolle der Vorgänge im System |**2-**Zusammenfassen mehrerer Interrupts zu Benachrichtigung (Interrup Coalescence): Reduziert Overhead für Interrupt-Verarbeitung, aber kann Latenz einzelner Anfragen deutlich erhöhen

weiterhin viele I/Os (Ordner-Inodes und –Daten lesen), aber danach kann der Cache genutzt werden -> 2 Ansätze:

Static Partitioning: Es wird im Arbeitsspeicher ein Cache mit fester Größe definiert (z.B. 10%). Dieser Speicherbereich kann nicht mehr für andere Aufgaben verwendet werden

Dynamic Partitioning: Die Pages der virtuellen Speicherverwaltung und Pages des Dateisystems bilden gemeinsam einen sogenannten ***Unified Page Cache***.

->Dynamischer Ansatz flexibler, aber komplizierter in Umsetzung ->Außerdem Konkurrenzkampf um Speicherressourcen.

Kosten der Speichervorgänge minimieren: -Schreibvorgänge verzögern und zusammenfassen (z.B. mehrere Änderungen an Inode) |-Dadurch werden manche Vorgänge komplett vermieden (Datei erstellen und löschen) |-Puffern ist Trade-Off zwischen Zuverlässigkeit und Geschwindigkeit (ca. 5-30s)