Zusammenfassung Betriebssysteme und Sicherheit

Henrik Tscherny

2. März 2021

Inhaltsverzeichnis

1 Betriebssysteme **2**
- 1.1 Aufgaben eines OS . . . 2
- 1.2 Prozesse . . . 2
- 1.3 Threads . . . 3
- 1.4 Adressraum . . . 4
- 1.5 Systemcalls . . . 5
- 1.6 Interrupts . . . 6
- 1.7 kritische Abschnitte . . . 7
- 1.8 Mutex/Locks . . . 8
  - 1.8.1 Semaphore . . . 8
  - 1.8.2 Deadlocks . . . 9
  - 1.8.3 Bedingungsvariablen . . . 9
- 1.9 Scheduling . . . 9
  - 1.9.1 Real-Time-Scheduling . . . 10

2 Speicherverwaltung **11**
- 2.1 Seitenersetzungsverfahren . . . 14
- 2.2 Seitenfehlerbehandlung . . . 16
- 2.3 Sicherheitskonzepte . . . 17
- 2.4 Dateisysteme . . . 18
  - 2.4.1 Aufbau . . . 18
  - 2.4.2 Links . . . 19
  - 2.4.3 Filedeskriptoren . . . 19
- 2.5 Inkonsistenten . . . 20

3 Sicherheit **21**
3.1 Schutz-Ziele 21
3.2 Verschlüsselung 22
3.3 Arten von Verschlüsselungsverfahren 22
3.4 RSA 22
3.5 Zufall 23

1 Betriebssysteme

1.1 Aufgaben eines OS

- Schnittstelle zwischen Software und Hardware
- Bereitstellen von CPU, Speicher, Ein-/Ausgabe und anderen Betriebsmitteln

1.2 Prozesse

Ein Prozess ist ein Programm welches ausgeführt wird. Die Bestandteile eines Prozesses sind:

- ein Programm
- mindestens ein Thread
- ein Adressraum
- Besitzer von Betriebsmitteln z.B. Netzwerkverbindungen, Speicherbereiche, geöffnete Dateien
- Nutzerzuordnung

Hintergrundprozesse nennt man auch deamons Folgende Ressourcen sind Prozess unabhängig:

- CPU-Cache
- Dateisystem
- physischer Speicher
- Netzwerkkarte

1.3 Threads

Ein Thread ist eine Aktivität die:

- ein sequenzielles Programm ausführt
- zu anderen Threads parallel läuft
- von OS bereitgestellt wird

Ein Thread kann sich in folgenden Zuständen befinden:

- aktiv: Thread wird ausgeführt, Pro CPU ist max ein Thread aktiv
- blockiert: mindestens eine nötige Ressource ist nicht vorhanden, die benötigt wird, damit die Ausführung starten kann (z.B. erwarten einer Eingabe, Freigabe von Betriebsmitteln)
- bereit: nicht blockiert aber nicht aktiv, Thread kann theoretisch los rechnen, ist aber noch nicht dran

Anforderungen an Threads sind:

- max ein Thread pro CPU
- ein aktiver Thread ist genau einer CPU zugehörig
- nur bereite Thread können die CPU erhalten
- Fairness: jeder Thread kommt wenn er rechen möchte auch dran
- das Wohlverhalten der Threads darf keine Voraussetzung bei der Implementierung sein

Threadumschaltung

- kooperativ
  - Threads rufen zuverlässig Umschaltoperationen auf
  - Threads geben freiwillig die Kontrolle ab
  - Verwendung heute teils noch in User-Level Threads
- präemtiv
  - Thread wird die CPU entzogen
  - Thread kann nichts dagegen tuen

- Thread Umschaltung kann zu jeder Zeit passieren

Threads teilen sich folgende Ressourcen:

- offene Datein
- globale Variablen
- Netzwerkkarte
- physischer Speicher
- PCB
- Virtueller Adressraum
- Dateisystem
- CPU-Cache

TCB

- Thread Zustand
- Kernel-Stack-Pointer
- CPU-State
- Verweis auf Adressraum
- Scheduling-Attribute

1.4 Adressraum

Als Teil eines Prozesses ist der Adressraum ein Ausschnitt des Hauptspeichers welcher vom Prozess verwaltet wird. Der Adressraum ist dabei wie folgt in mehrere Abschnitte untergliedert:

Kern Von den höchsten Adressen startend beginnt der Teil des Adressraums welcher einen Teil des Kernel enthält. Dazu zählen u.a. eine Liste aller Prozesse und eine Tabelle der offenen Dateien. Dies ist z.B. für die Durchführung von Systemcalls nötig. Ein im Usermode laufender Prozess kann nicht auf dieses Segment zugreifen.

Stack Beim Stack handelt es sich um einen Kellerspeicher. Der Stack dient dem Speichern von lokalen Variablen und Funktionsparametern. Auch werden Rücksprungadressen auf dem Stack gespeichert. Da Funktionen immer in der Reihenfolge verlassen werden wie sie betreten wurden, ist hierbei der Kellerspeicher als LIFO Speicher sehr hilfreich. Der Stack wächst von hohen Adressen hin zu niedrigeren Adressen. Das Oberste Element des Stacks wird mit den Stackpointer(SP) markiert.

Heap Unter den Stack folgt der Heap. Dieser wächst entgegen der Richtung des Stack von den niedrigen Adressen zu höheren Adressen. Auf dem Heap werden alle zu Laufzeit erstellten Speicherbereiche, wie etwa durch malloc() gelagert. Der Heap wird von allen Threads und geteilten Bibliotheken eines Prozesses genutzt.

BSS Das folgende BSS Segment enthält alle globalen Variablen welche nicht initialisiert wurden, d.h. welch keine Wertzuweisung haben.

Data Im Daten Segment befinden sich globale Variablen welche Initialisiert wurden

Text Ganz unten im Adressraum befindet sich das Textsegment. Das Textsegment enthält den auszuführenden Programmcode und ist Read-Only und von fester Größe. Oft wird am unteren Ende etwas Platz gelassen um nicht initialisierte Pointer zu erkennen.

1.5 Systemcalls

Systemcalls sind spezielle Befehle welche genutzt werden können um Funktionen des OS-Kernels zu benutzen.

- der Kernmodus wird eingeschaltet
- es wird auf den Kern Stack umgeschaltet
- es wir an eine andere Stelle verzweigt
- der Kern führt die dortige Operation aus

Beispiele für Syscalls:

- x = fork(): Erstellt eine exakte Kopie des Aufrufers inc. Adressraum, Filedeskriptoren usw.

  - x == 0: im Child Process
  - x < 0: Fehler
  - x > 0: Parentprocess , enthält ID des Childs

- exit(status): beendet Prozess. Wenn Kindprozess, dann bleibt dieser noch als Zombie bestehen bis der Parent wait() ausführt
- s = wait(pid): wartet auf die Beendigung des Childs mit pid, wenn pid == -1, warte auf beliebiges Child, s ist der exit(status) des Kindes
- s = exec(file, arg, env): ersetzt Speicherinhalt des Prozesses durch den Inhalt von File, schreibt Argumente und Umgebungsvariablen an Anfang des Kellersegments

1.6 Interrupts

- unterbrechen den Ablauf eines aktiven Threads an einer beliebigen Stelle
- asynchron: Interrupts, synchron Exceptions
- Auslöser: IO-Geräte, interne Timer
- Exceptions: Seitenfehlern, fehlerhaften Instruktionen, explizites Auslösen durch Syscalls
- Timerinterrupts können vom Kern mit speziellen Syscalls deaktiviert/aktiviert werden (cli (aus), sti (an))

ein IO-Interrupt läuft wie folgt ab:

- IO-Controller löst Interrupt aus
- CPU erkennt Auftreten des Interrupts zwischen den Befehlen
- CPU schalten in der Kernel-Mode und schaltet auf den Kernel-Stack welcher im TCB vermerkt ist
- CPU sichert dort User-Stack-Pointer, User-Programm-Counter, Flags, usw
- CPU lädt Kern-Programm-Counter aus Interrupt Description Table (IDT)
- Fortsetzung des Codes im OS-Kern
- Instruktion iret stellt Zustand vor dem Syscall für den Thread wieder her
- Thread fährt mit der Ausführung fort

1.7 kritische Abschnitte

Anforderungen an einen kritischen Abschnitt:

- es darf sich maximal ein Thread zur gleichen Zeit in ihm befinden
- jeder Thread der ihn betreten möchte bekommt irgendwann auch die Gelegenheit dazu
- es dürfen keine Annahmen bzgl. der Reigenfolge, relativen Geschwindigkeit der Threads gemacht werden

Race-Condition

Eine Race-Condition tritt auf, wenn mehrere Dinge (z.B. Thread) auf die gleiche Ressource zugreifen und so das Ergebnis einer Operation davon abhängig ist, welcher Thread als erstes an der Reihe ist und wie die weitere Abarbeitungsreihenfolge der Threads abläuft (z.B. doppelte Abbuchen eines Kontos)

Lösung nach Peterson

- jeder Thread bekundet sein Interesse einen kritischen Abschnitt zu betreten
- dafür benutzt man eine List $interested[self.id] = true$
- nach dem Bekunden des Interesses setzt der Thread eine turn variable auf eine andere ID als seine Eigene $turn \neq self.id$
- es wird dann eine Endlosschleife betreten solange der andere Partner das Interesse bekundet und die ID des Partners auch dran ist $while(interested[other.id]\ and\ turn == other.id)$
- nach dem Verlassen der kritischen Sektion wird das eigene Interesse entbekundet $interested[self.id] = false$
- Node: funktioniert nicht für OS mit weak memory consistency

andere Lösungsversuche:

- Unterbrechungssperre: ausschalten der Timerinterrupts
  - nur Kern kann Timerinterrupts sperren
  - deaktiviert nur den Timer des eigenen Kerns, die Timer anderer Kerne laufen weiter $\rightarrow$ nur für einen Core möglich

- nicht Unterbrechbare Instruktionen
  - da jede Instruktion selbst aus mehreren μOps besteht wird das Problem nur weiter nach unten gereicht
- atomare Instruktionen
  - Speicherzelle wird Hardwareseitig gesperrt und ist Hardwareseitig ganz oder garnicht ausführbar
  - funktioniert
  - früher: Bus wird gesperrt, heute: Cache-line wird gesperrt

1.8 Mutex/Locks

- bussy waiting
- hohe Busbelastung
- ein Thread kann verhungern
- nicht für Thread auf derselben CPU
- nur für kurze kritische Abschnitte geeignet

1.8.1 Semaphore

- Zähler
- Warteschlange
- Operationen up() und down() manipulieren den Zähler
- können von verschiedenen Thread verwendet werden
- kein Bussy-Waiting, sondern blockieren in der Warteschlange
- sleep(q) blockiert Aufrufer und fügt in in WS ein
- wakeup(q) weckt den ältesten Thread in WS auf

1.8.2 Deadlocks

- ein Deadlock ist eine Situation wobei mehrere Thread jeweil zyklisch aufeinander warten
- es kommt zu einem Stillstand, da jeder auf einen anderen wartet
- kann durch Locks/Semaphore ausgelöst werden
- Notwendige Bedingungen sind:
  - gegenseitiger Ausschluss bei der Benutzung von BM
  - Nachfordern von BM
  - keine Verdrängung möglich
  - zyklische Wartesituation
- Hinreichend: Konjunktion der 4 Notwendigen

1.8.3 Bedingungsvariablen

- Warteschlange
- wait() und signal()
- Bedingung frei implementierbar
- Test der Bedingung muss mit Lock geschützt werden
- Lock wird mit wait() atomar freigegeben und beim Aufwachen wieder gesperrt

1.9 Scheduling

- Optimale Schedule: Es gibt keine bessere Reihenfolge welche eine geringere Zeit benötigt um alle Jobs zu bearbeiten
- Wartezeit: Zeit eines Jobs bevor er anfangen kann zu Rechnen, d.h. Ausführungszeit der Jobs davor + Lücken
- Verweilzeit: Zeit die ein Job bis zur Beendigung braucht, d.h. Wartezeit + eigene Ausführungszeit

Shortest Processing Time (SPT)

- nimmt immer den kürzesten Job und reiht diesen als erstes ein
- Optimal wenn es keine Abhängigkeiten zwischen Aufgaben gibt
- gut um gesamt Wartezeit zu minimieren

Longest Processing Time (LPT)

- reiht den Job mit der längsten Ausführungszeit zuerst ein
- LPT ist optimal, sofern es nur eine CPU gibt (trivial)
- gut um gesamt Bearbeitungszeit zu minimieren

Round-Robin

- Jeder Job bekommt reihum jeweils ein Timeslice mit Größe Q
- es gibt n Jobs mit Ausführungszeit T
- ein Zyklus sei ein Abschnitt in welchem jeder Job einmal gerechnet hat
- Wartezeit für einen Zyklus $t_{w_i} = Q(i-1) + (\frac{T}{Q} - 1)(n-1)$
- durchschnittliche Gesamtwartezeit $\frac{1}{n} \sum_{i=1}^{n} t_{w_i}$

1.9.1 Real-Time-Scheduling

- eine Menge von Jobs $j \in J$ mit besonderen Eigenschaften
- Hyperperiode: $kgV(p_j), j \in J$, Punkt an dem der Schedule sich wiederholt
- Auslastungsgrad: $\eta = \sum_{i_1}^{n} \frac{e_i}{p_i}$
- jeder Job hat:
  - Ausführungszeit e
  - Deadline d (hard oder soft)
  - Periode p
  - keine Abhängigkeiten *(hier)
  - unterbrechbar
  - kein Scheduling Overhead *(nur theoretisch)

Admission

- Zeigt ob der Task tatsächlich einplanbar ist
- Einplanbarkeit kann durch das Admission-Kriterium überprüft werden

Rate Monotonic Scheduling (RMS)

- Jeder Job besitzt eine feste Priorität
- Jobs mit kürzester Periode erhält höchste Priorität
- Optimal bzgl. Einplanbarkeit unter statischen Prioritäten
- geringer Overhead
- schlechte Auslastung
- Admission: Einplanbar wenn $\eta \leq n(\sqrt[n]{2} - 1), n = |J|$

Earliest Deadline First (EDF)

- benutzt dynamische Prioritäten
- Job mit nächst gelegener Deadline bekommt höchste Priorität
- Optimal bzgl. Einplanbarkeit unter dynamischen Prioritäten
- großer Overhead
- höhere Auslastung
- Admission: Einplanbar gdw. $\eta \leq 1$

2 Speicherverwaltung

Aufbau

- Seite: virtuelle Abbildung eines Teils des theoretisch möglichen Hauptspeichers
- Rahmen: Zu einer Seite korrespondierender Speicherbereich im Hauptspeicher
- zu einer Seite gehört, sofern das present-bit gesetzt ist mindestens ein Rahmen, es können mehrere Seiten auf den selben Rahmen zeigen

- Block: Speichersegment auf der Festplatte (persistent)
- Seiten werden in der Seitentabelle gespeichert
- die Seitentabelle speichert auch die Rechte für die jeweilige Speicherregion
- Es können Seitenverzeichnisse benutzt werden welche Verweise auf Seitentabellen enthalten
- durch die Hierarchie können große Menge an Seiten auf einmal Verwaltet werden (z.B. als nicht Präsent oder nur lesbar markiert werden)
- die MMU ist für die Umrechnung der virtuelle Adressen in die physischen Zuständig
- der TLB speichert einige Übersetzungen für virtuelle in physische Adressen, für häufig verwendete Adressen

Statische Segmentierung

Jedes Programm erhält die gleiche Menge von Speicher Folgende Nachteile entstehen durch das nicht verwenden von virtuellem Speicher:

- Es werden dedizierte Register (Basis und Limit-Register) benötigen zum Schutz vor unerlaubten Zugriffen
- Reallocation: Je nachdem in welchem Segment sich das Programm befindet sind die Adressen anders (Lösung $\rightarrow$ Base-Register)
- Fragmentierung des Speichers, da Programme viel zu Große Segmente erhalten $\rightarrow$ viel verschwendeter Speicherplatz

Dynamische Segmentierung

First-Fit:

- platziert die Anfrage in die erste Lücke in welche das Programm passt
- Beginnt Suche immer am Anfang des Speichers
- sehr einfaches Verfahren
- erzeugt starte Fragmentierung des Speichers, besonders am Anfang des Speichers

Next-Fit:

- First-Fit, beginnt jedoch die Suche am Ende des zuletzt platzierten Stücks
- einfaches Verfahren
- erzeugt Fragmentierung wie bei First-Fit, jedoch über den gesamten Speicher verteilt
- bessere Verteilung der Fragmentierung

Best-Fit:

- platziert die Anfrage in beste Lücke, d.h. die welche die geringste externe Fragmentierung erzeugt $\rightarrow$ Lücke dessen Größe am nächsten der Größte der Anfrage entspricht
- hoher Suchaufwand
- besseres Einlagerungsverhalten
- jedoch möglich, dass Best-Fit schlechter ist als First-Fit ist, allgemein jedoch besser
-

Worst-Fit

- nimmt die Lücke welche die Größte Fragmentierung erzeugt $\rightarrow$ die Lücke wessen größte am weitesten von der angefragten Größte abweicht
- hoher Suchaufwand
- etwas schlechter als Best-Fit
- sogt für gleichgroße Lücken

Buddy-Verfahren

- der Speicher wird in Blöcke der Größe 2^k zerlegt
- der angeforderte Speicher wird zur nächsten Zweierpotenz aufgerundet
- gibt es keinen Block dieser Größer so wird die nächst Größere Zweierpotenz gesucht, diese zweigeteilt und eine Hälfte dem Prozess zugewiesen
- geringer Suchaufwand

- einfach Implementierung
- keine Hardwareunterstützung nötig
- erzeugt interne und externe Fragmentierung
- schlechte Einlagerung als Best-Fit
- Worst-Case wenn angefragte Speichergrößen $2^n + 1$ sind (wegen Aufrundung)

2.1 Seitenersetzungsverfahren

- OPT
  - Basiert auf Wissen von der Zukunft $\rightarrow$ nicht implementierbar
  - Dient als theoretische untere Schranke
- FIFO
  - Nutzt kein Wissen über Referenzverhalten eines Programms (Lokalität)
  - sehr einfach implementierbar
  - durchschnittlich schlechtes Verhalten
  - Auftreten der Beladeschen Anomalie (mehr Seitenfehler bei mehr Speicher)
- LRU/LFU
  - Gute Näherung an OPT
  - aufwändige Implementierung (komplettes Referenzverhalten, volle Suche)
- Clock-Algorithmus
  - Annäherung an LRU-Verfahren
  - einfache Implementierung
  - Nicht fair zwischen mehreren Prozessen, da ein Prozess komplett den physischen Speicher übernehmen kann
  - MMU vermerkt Zugriffe auf Seiten in Accessed-Bit der Seitentabelle
  - Seiten ohne Accessed-Bit werden verdrängt

  - Funktionsweise
    * Ist Access-Bit an Zeigerstelle gesetzt ?
    * Wenn JA: Bit in Seitentabelle zurücksetzen, gehe zum nächsten Frame
    * Wenn NEIN: Kandidat gefunden gefunden, tausche Seiten aus, gehe zum nächsten Frame

- Arbeitsmengenmodell
  - Nutzt Lokalität für Ersetzungsentscheidungen
  - Zugriffe werden für jeden Prozess einzeln aufgezeichnet (Arbeitsmengen)
  - Alle Seiten innerhalb der letzten n Zugriffe (Fenstergröße) werden nicht verdrängt
  - Berücksichtigt Lokalität
  - Fair zwischen mehreren Prozessen, da bevorzugt lokale Verdrängung
  - aufwändige Implementierung (speichern und updaten der AMs, jeder Zugriff muss aufgezeichnet werden)
  - Funktionsweise
    * Bei jedem Zugriff werden die AM aktualisiert, (Liste der letzten n Zugriffe, keine Doppelnennung bei mehrfachen Zugriffen)
    * Kacheln welche nicht mehr in der AM sind werden als ersetzbar markiert
    * Verdränge zuerst eigene Seiten (lokale Verdrängung), sonnst Seiten von anderen (globale Verdrängung)

- Second Chance
  - jede Seite erhält eine referenced Flag
  - wird eine Seite referenziert so wird diese gesetzt
  - wird eine Kachel benötigt, so wird vom Anfang der Liste durchsucht
  - Kachel hat Flag: Flag wird gelöscht, gehe zum nächstem Element
  - Kache hat keine Flag, Kachel wird ausgetauscht
  - kann mit dem Clock-Algorithmus kombiniert werden
  - enorm besser als FIFO

- Aging

  - für jede Seite für eine Liste (z.B. aus 8 Bits) gespeichert
  - bei jedem Tick wird an den Anfang ein Bit angesetzt und das hinterste Bit herausgeschoben
  - Seite hat die Flag: 1 wird vorn angefügt, Flag wird gecleared
  - Seite het keine Flag: 0 wird vorn angefügt
  - wird eine neue Kachel benötigt, so wird die Kachel eingelagert mit dem kleinsten Alter (Alter ist die Zahl welche sich aus den 8 Bits ergibt)
  - effizienter Algorithmus

2.2 Seitenfehlerbehandlung

- Echte Fehler
  - Zugriffe auf ungültige Speicherbereiche
  - Prozess wird SIGSEGV zugestellt
  - Prozess wird (im Normalfall) beendet
- Reparable Fehler
  - Zugriff gültig, aber Seite ausgelagert
  - wird transparent im OS behandelt
  - OS lagert Seite ein
  - Programm wird normal fortgesetzt
- genereller Ablauf:
  - Seitenfehler tritt auf
  - finde Objekt und Seite (Offset Behandlung)
  - Suche ob Kachel schon vorhanden ist (von einem anderen Thread bspw.)
  - Falls Ja: Seite in Seitentabelle eintragen, done
  - Falls Nein: suche nach freien Kacheln
  - Es gibt eine freie Kachel: Inhalt lesen, Seite eintragen
  - Es gibt keine freie Kachel: verdrängen
    * Aus allen beteiligen Seitentabellen austragen
    * Zu welchem Objekt gehört die verdrängte Kachel
    * Inhalt sichern
  - neuen Inhalt lesen, Seite eintragen

2.3 Sicherheitskonzepte

- Objekt: Dateien mit Besitzer und Gruppe
- Subjekt: Prozesse, die als ein Nutzer sowie all seinen Gruppen gestartet werden
- Rechte: lesen, schreiben, ausführen (rwx)

ALC

- darf von Objekterzeuger gesetzt werden
- bei jedem Zugriff wird beim Objekt auf der Basis der ID des Aufrufers dessen Berechtigung überprüft
- benutzt in reduimentären FS
- Aufbau: Datei [Besitzer] [Gruppen] [Rechte]
- geringe Ausdrucksmöglichkeiten
- verfügt u.u. noch über ein set-uid Bit, wenn gesetzt wird das Programm zusätzlich zu den Rechten des Nutzers mit den Rechten des Besitzers Ausgeführt
- set-uid dient dazu Ressourcen vor unprivilegierte Zugriffen zu schützen

Capabilities

- bei jedem Zugriff wird etwas geprüft, was Subjekte besitzung und bei Bedarf weitergeben können
- Beschränkung durch die initiale Vergabe der Caps
- Subjekte können Rechte gezielt weitergeben
- Erzeuger eines Objekt hat zunächst alle Rechte und gibt diese weiter
- Überprüfung der Caps durch vertrauenswürdige unumgängliche Einheiten (z.B. Kern oder spezielle Dienste)

Mandatory Access Control (MAC)

- bei jedem Zugriff werden Regen ausgewertet
- Subjekte und Objekte haben labels
- Entscheidung über Zugriffe anhand von Regeln
- z.B. Multilevel Security durch verschiedene Sicherheitsebenen

2.4 Dateisysteme

2.4.1 Aufbau

Superblock

- Ankerpunkt für das FS
- Root I-Node
- Allocation Bitmap (Zeigt welche physische Blöcke verwendet werden)
- I-Node Bitmap (Zeigt welche I-Nodes im Block vergeben sind)

Root I-Node

- Rechte
- Größte
- uid, gid
- Verzeichnisblock

I-Node Blöcke

- I-Nodes
- I-Nodes verweisen wieder auf Verzeichnisblöcke

Verzeichnisblock

- Verweis auf Datenblöcke
- I-Nodes mit weiteren Verzeichnisblöcken
- evl. Verweise auf Indirection Blöcke, diese zeigen wiederum auf Datenblöcke

2.4.2 Links

Hardlink

- Verweis im einem Verzeichnisblock
- Zeigt auf eine I-Node
- d.h. mehrere Verzeichniseinträge zeigen auf den selben Datei I-Node
- dürfen nicht auf Verzeichnisse zeigen, da so Schleifen entstehen können
- Eintrag im Verzeichnisblock kann gelöscht werden, es wird dabei lediglich der Link-Counter reduziert, die Datei ist jedoch immernoch über den Hardlink (der ja immernoch auf diese Datei zeigt) erreichbar
- kann nicht über FS Grenzen hinausgehen
- Da der Hardlink auf die I-Node direkt zeigt, funktioniert er auch wenn die Datei verschoben werden sollte

Softlink/Symbolic Link

- ganz normale Datei eines I-Nodes
- Datei enthält einen Pfad
- OS löst Pfad in der Datei nochmal von Anfang auf
- Wird der Verweis auf die Datei gelöscht so wird dabei der Link-Counter auf 0 reduziert da keine Links zu der Datei mehr existieren werden die Daten freigegeben $\rightarrow$ der Softlink zeigt ins nichts
- kann über FS Grenzen hinaus verlinken (z.B. Netzwerkpfade)
- können auch auf Verzeichnisse verweisen, das OS übernimmt den loop-check

2.4.3 Filedeskriptoren

- Werden in einer von Kernel verwalteten Liste gespeicherts
- ein Eintrag pro offener Datei
- die offenen Dateien der Prozesse werden in der Open File Table gehalten
- die Einträge der OFT zeigen auf I-Nodes von Dateien

- Zugriff über die Nummer des Eintrags
- 0: Std-In
- 1: Std-Out
- 2: Sdt-Err

2.5 Inkonsistenten

kritisch

- Verweise ohne hinterlegte Daten
- Verweise auf Blöcke oder I-Nodes, die noch nicht angelegt worden/nicht mehr existieren z.B. wenn mein erzeugen/löschen einer Datei der Strom ausfällt
- Verwendung von I-Nodes welche noch nicht als belegt markiert wurden, wenn z.B. ein Ausfall geschieht bevor ein Eintrag in der Allocation Bitmap erzeugt werden konnte
- möglicher Verlust des Dateisystems

unkritisch

- angelegte Daten ohne Verweise
- Nur Datenverlust, nicht aber das Dateisystem
- Blöcke sind als Belegt markiert, jedoch sind jedoch ohne Verwendung, Allocation Bitmap ist bereits belegt, jedoch gibt es keine Daten dazu
- kann mit einen FS check wiederherstellbar

Vermeidung durch synchrones Schreiben

- schreibe zuerst die Allocation Bitmap
- schreibe die Datenblöcke
- führe eine write-barrier aus (sogt dafür, dass alle Daten auf der Platte persistent werden bevor weiter gemacht wird)
- schreibe den zugehörigen I-Node Block

- schreibe den Superblock (egal wann da nur für Statistik)
- Essence: Schreibe immer erst den Verweis und dann die Daten
- nur unkritische Inkonsistenzen möglich
- write-barrier nach jedem Schreiben von Daten

Journaling

- es gibt einen dedizierten Bereich auf der Festplatte (das Journal)
- statt Änderungen inplace zu machen, werden die Änderungen erst in einem Journaleintrag geschrieben
- es gibt Metadata (Nur Verweise) und Full Journaling (Verweise und Datenblöcke)
- bei Absturz kann man dann in das Journal schauen um Inkonsistenzen zu finden
- Vorteil: Weniger write-barriers (eine für in da Journal schreiben, eine nach dem schreiben der Transactionblöcken, eine nach dem inplace schreiben)
- Vorteil: Compound Transactions: zusammenfassen von Änderungen
- Nachteil: benötigt mehr Platz
- Nachteil: mehr Schreiboperationen

3 Sicherheit

3.1 Schutz-Ziele

- (C) Vertraulichkeit: Daten nur von bestimmten Personen lesbar/modifizierbar
- (I) Integrität: Daten dürfen nicht unbemerkt verändert werden, Veränderungen müssen nachvollziehbar sein
- (A) Verfügbarkeit: Zugriff auf Daten muss ïrgendwannmöglich sein
- C: Durch Verschlüsselungssysteme
- I: Durch Authentifizierungssysteme
- A: Durch Ausfallsicherungssysteme

3.2 Verschlüsselung

Kerkhoff'sches Prinziep

Die Sicherheit eines Kryptographischen Systems sollte nicht von der Implementierung dessen abhängig sein, d.h. der Algorithmus muss auch sicher sein, wenn er open-source ist

3.3 Arten von Verschlüsselungsverfahren

symmetrische Verschlüsselung: Selber Key für enc(x) und dec(x)

sym. V. sind schnell aber schwerer Schlüsselaustausch

asymmetrische Verschlüsselung

asym. V. sind langsam aber einfacher Schlüsselaustausch

Kombination von asym. V. und sym. V. als hybrid

asym. V zum Austausch des Schlüssels für sym. V. (z.B. TLS)

3.4 RSA

- Faktorisierung ist NP-Hard und $P \neq NP$
- Key Generation
  - wähle zwei Primzahlen p und q ($|p| \approx |q|$)
  - rechne $n = p \cdot q$ es gilt außerdem $\varphi(n) = (p-1) \cdot (q-1)$
  - wähle ein $c \in (a, \varphi(n)$ mit $ggT(c, \varphi(n)) = 1$
  - berechne ein $d \in (1, \varphi(n))$ mit $c \cdot d \equiv 1 \ mod \ \varphi(n)$
  - public key: (n,c)
  - private key: (n, d)
- De/Encryption
  - Nachricht: x
  - Encryption: $x^c \equiv y \ mod \ n$
  - Decryption: $y^d \equiv x \ mod \ n$
  - $x, y \in [0, n]$
- Note: Nachweis dass c und d ein Keypair sind mittels $c \cdot d \equiv 1 \ mod \ \varphi(n)$

3.5 Zufall

Erzeugung von Entropie durch:

- Jitter von Interrupts
- Abstände zwischen Interrupts von IO-Geräte
- Rauschen auf Mikrophonen oder Pins
- Hardware Random Number Generator (HRNG)

Schlechter Zufall:

- $n_1 = p_1 \cdot q_1$
- $n_2 = p_2 \cdot q_2$
- Sei $p_1 = q_2$
- n_1 und n_2 sind so leicht findbar
- $ggT(n_1, n_2)$ ergibt den einen Primfaktor der für beide gleich ist, der andere lässt sich dann aus $\frac{n}{p}$ einfach errechnen
- einfach angreifbar durch paarweises durchtesten von Publickey listen $\rightarrow$ public keys öfter über Zeit immer mal wieder neu generieren