Zusammenfassung: Betriebssysteme

Grundlagen

Betriebssystem-Arten

- Mainframes: High End Systeme, z. B. z/OS
- Server-BS: Serversysteme für Client-/Server-Architekturen z. B. Linux, Unix, Solaris
- PC-BS: z. B. Linux , Windows, Mac OS
- Echtzeit-BS: garantierte Reaktionszeiten z. B. VxWorks, QNX, OSEK
- Embedded Systems und Handheld-Systeme: kleine Systeme für PDAs, Gerätesteuerung, z. B. Android, PALM OS, Symbian OS, iOS, Windows CE
- Smart Card BS: für Kreditkarten, kleine Programme

Anforderungen und Aufgaben

- entlastet Anwender/Entwickler von Details der Hardware
- kapseln den Zugriff auf Betriebsmittel → Zugriff nur über Betriebssystemfunktion (Systemdienst), "Virtuelle Maschine" über Hardware
- Dateiverwaltung, Prozess und Speicherverwaltung, Hardwareabstraktion, Geräteverwaltung
- Betriebsmittelverwaltung (Hard-/Softwareressourcen), z. B. Prozesse, CPU, Dateien, ...

Architekturen

- Monolitisch: alle Module haben Zugriff auf einen Adressraum und teilen sich diesen; Kernel, in dem nicht nur Funktionen zu Speicher- und Prozessverwaltung und zur Kommunikation zwischen den Prozessen, sondern auch Treiber für die Hardwarekomponenten und möglicherweise weitere Funktionen direkt eingebaut sind.
  - Nachteile: Kernel fehleranfälliger, da der Teil, der abgestürzt ist, nicht einfach neu gestartet werden kann, sondern sogar einen Absturz des gesamten Systems nach sich ziehen kann; bei Änderungen muss kompletter Kernel neu übersetzt werden
  - Vorteile: entfällt aufwändige Kommunikation zwischen den einzelnen Teilen des BS
- Schichten: verschiedenen Betriebssystemkomponenten wie Schalen aufeinander
- aufgebaut
  - Vorteile: flexibler und übersichtlicher als monolithischer Kern, leichter zu konfigurieren, möglichst wenig Hardware-Abhängigkeit
- Mikrokern: nur Funktinen zur Speicher- und Prozessverwaltung, Grundfunktionen zur Synchronisation, Kommunikation; alles weitere wird in Serverprozesse ausgelagert die mit nachfragenden Programmen (Clients) kommunizieren oder als Programmbibliothek, welche von nachfragenden Programmen eingebunden wird (im Benutzermodus implementiert)
  - Vorteil: absturzsicherer, klein/schlank, einzelne Bestandteile beliebig austauschbar, elementarer Kern ermöglicht verschiedene, darauf aufsetzende Betriebssysteme
  - Nachteil: Performanceverschlechterung

Unix/Linux-Schichtenmodell (klassisch monolithisch)

Anwendungsprogramme (vi, sh, sed, …), C-Library, weitere Libs
→ Usermodus

System Call Interface (Speicherverwaltung, Geräteverwaltung, Dateiverwaltung, Prozessverwaltung)
→ Kernelmodus

Windows-Schichten

Environment-Subsystems: Umgebung für Anwendungsprogramme
POSIX- und OS2-Subsystem: nach Windows 2000 nicht mehr verfügbar
Executive: stellt Systemcall-API bereit
Microkernel: Multiprozessor-Scheduling, Thread-Scheduling, Interrupt-Handling
HAL: kapselt Hardware

Betriebsarten

- Einprogramm (Singletasking): nur ein (Teil)-Programm aktiv, das sämtliche Betriebsmittel erhält
- Mehrprogramm (Multitasking): mehrere Programme aktiv, benötigte Betriebsmittel abwechselnd zugeteilt nach Prioritäten/Zeitscheibenverfahren

- Stapelverarbeitung: zu bearbeitender Job muss für Bearbeitung vollständig sein; Jobs werden in Warteschlange verwaltet und nach definierter Strategie abgearbeitet
- interaktive Verarbeitung: Auftrag muss nicht vollständig sein; permanente Kommunikation des Nutzers mit BS über UI
- verteilt
  - Teilhaberbetrieb: mehrere Anwender arbeiten gleichzeitig am selben Rechner mit demselben Programm (Transaktionsmonitor, teilt Prozesse und sonstige Betriebsmittel zu) → System führt Anforderungen in Transaktionen aus
  - Teilnehmerbetrieb: mehrere Anwender arbeiten mit zentralem Rechner, aber mit unterschiedlichen, voneinander unabhängigen Programmen und Daten; Anwender sieht Rechner wie eigenen; jeder enthält Prozess und sonstiger Betriebsmittel zugeteilt

Systemcalls

- Anwendungsprogramme nutzen Dienste des Betriebssystems, die über sog. Systemcalls aufgerufen werden
- wohldefinierte Einsprungpunkte ins Betriebssystem
- spezieller Aufrufmechanismus für Systemcall
  - Software-Interrupt (als **Trap** bezeichnet) oder Supervisor Call
  - Vorteil: Anwendungsprogramme muss Adressen der Systemroutinen nicht kennen
- Alle Systemcalls zusammen bilden die Schnittstelle der Anwendungsprogramme zum Betriebssystemkern
- Zugang zu Systemcalls wird meist in Bibliotheken bereitgestellt
- Standards: POSIX, Win32-API

Befehlsfolge

- Softwareinterrupt: Parameter auf Stack legen, PC ← Adresse (Systemroutine)
- Umschalten auf Kernel Mode
- Betriebssystemkern

Hardware

Interrupts/Traps: Unterbrechungen

- ermöglichen auf Polling/Busy-Waiting/aktives Warten *zu verzichten*
  - Polling: ständige CPU Last durch zyklische Abfrage von Ereignisquellen
- Gründe für Interrupts: Betriebssystembedingungen, asynchrone Ereignisse
- Verursacher: Hardware oder Software
- Abschaltung/Maskierung möglich, aber nur für sehr kurze Zeit sinnvoll
- *Synchrone* Interrupts: von CPU ausgelöste Ausnahmen (für laufendes Programm gedacht), z. B. Division durch 0 oder Speicherzugriffsverletzung
  - Exceptions: Faults, Traps
  - Systemcalls
- *Asynchrone* Interrupts: unabhängig davon, was System gerade ausführt, z. B. Netzwerkadapter meldet ankommende Nachricht, Plattenspeicher meldet Zustellung eines Blocks
- Behandlung: Ablauf einer Interrupt Service Routine
  - Interrupt Request (IRQ) wird vom Gerät gesendet und identifiziert Gerät
  - Interupt-Controller (IRQ) ordnet Geräte IRQ zu in Interrupt-Vector-Tabelle (IVT)
  - Am Ende eines Maschinenbefehls wird geprüft, ob Interrupt anliegt
  - Schreibe Rücksprungadresse auf Stack oder in dafür vorgesehenes Register
  - Schreibe Registerinhalte auf Stack
  - Bearbeite Interrupt
  - Geh wieder zurück zu vorheriger Bearbeitung
- Prioritäten: je niedriger Zahl des Levels, desto höher Priorität; Interrupt mit höherer Priorität kann einen mit niedrigerer unterbrechen
- Timer Interrupt/Clock Intervall → meldet nach bestimmten Zeitintervall einen Interrupt → CPU-Scheduling

CPU

- User-Modus: Ablaufmodus für Anwendungsprogramme; kein Zugriff auf Kernel-spezifische Code- und Datenbereiche
- Kernel-Modus: privilegierter Modus; dient Ausführung der Programmteile des Kernels; Schutz von Datenstrukturen des Kernels
- Umschaltung erfolgt über spezielle Maschinenbefehle
- Statusregister: aktueller Modus steht in Statusregister

Shell-Programmierung

Variablenzuweisungen: `var1=wert1 var2=wert2`
Ausgabe aller definierten Variablen mit `set`
Anzahl der Parameter: `$#`, alle Parameter: `$*` bzw. `$@`, Rückgabewert des letzten Aufrufs: `$?`, Prozessnummer der Shell: `$$`, Prozessnummer des letzten Hintergrundaufrufs: `$!`, Parameter beginnend mit -: `$-`, letztes Argument des letzten Aufrufs: `$_`

Kontext für Berechnungen: `expr … ` oder `$(( … ))` oder `$[ … ]`

Ausgabe in Datei: `>` bzw. `>>` Eingabe aus Datei: `<` Pipe: `|`

Kontrollstrukturen:

```
if …; then … fi
case $var in cond1) …;; cond2) …;; *) …;;esac
for var in …; do … done
for (( i = 1; i <= $max; i++ )); do … done
while …; do … done      until …; do … done
```

Prozesse/Threads

Prozesse

Definition: Prozess ist Programm während der Ausführung inklusive Dateien und Registerinhalten.

- **Virtueller Prozessor**: Betriebssystem ordnet im Multiprogramming jedem Prozess einen virtuellen Prozessor zu
- echte Parallelarbeit, falls jedem virtuellen Prozessor ein realer zugeordnet wird
- quasi parallel: jeder realer Prozessor ist zu einer Zeit immer nur einem virtuellen Prozessor zugeordnet und es gibt Prozessumschaltungen (Kontextwechsel)
- Prozess wird mit Mitteln des Betriebssystems erzeugt
  - realen Prozessor, Hauptspeicher und weitere Ressourcen zuordnen
  - Programmcode und Daten in Speicher laden
  - Prozesskontext laden und Prozess starten
- Gründe für Beenden: Normaler exit, Error exit (vom Programmierer gewünscht, fataler error), durch anderen Prozess beendet
- **verschiedene Zustände**: bereit (Prozess wartet bis Prozessor zugeteilt wird), aktiv, blockiert (Prozess wartet auf Betriebsmittel, z. B. Festplatte), beendet
- Betriebssystem verwaltet Prozesstabelle: Information, die die Prozessverwaltung für Prozesse benötigt, wird in einer Tabelle bzw. mehreren Tabellen/Listen verwaltet
- Ein Eintrag in der Prozesstabelle wird auch als *PCB* (Process Control Block) bezeichnet
  - Enthält unter anderem: Programmzähler, Prozesszustand, Priorität, verbrauchte Prozesszeit seit Start, Prozessnummer, Elternprozess, zugeordnete Betriebsmittel (z. B. Dateien)
  - UNIX besitzt baumartige Prozessstruktur (Prozesshierarchie)
    - neuer Prozess wird mit fork() erzeugt
    - Rückgabewert: PID im Parent, 0 im Kindprozess
  - Kindprozesse erben Umgebung von Elternprozess als *Kopie*
    alle offenen Dateien und Netzwerkverbindungen, Umgebungsvariablen, aktuelles Arbeitsverzeichnis, Datenbereiche, Codebereichte
  - durch execve() kann im Kindprozess neues Programm geladen werden
- „Zombie-Prozess“ unter UNIX: Prozess ist beendet, aber Elternprozess benötigt noch PID (z. B. waitpid())

Threads

„Leichtgewichtige Prozesse“: erlauben Parallelisierung der Prozessarbeit

Unterschied zu Prozessen: Jeder Prozess hat eigenen Adressraum und kann mehrere Threads enthalten; Threads teilen sich einen Adressraum.

Vorteile: weniger Verwaltungsaufwand, Kontextwechsel geht schneller, sinnvoll bei Systemen mit mehreren CPUs
Nachteile: Anwendungsprogrammierer benötigen Kenntnisse

Implementierungen

- JVM unterstützt Threads; Verwendung/Verwaltung über Klasse „Thread“
  - implementiert Interface „Runnable“
  - innerhalb von start() wird automatisch run() aufgerufen
  - join() wartet bis sich Thread beendet
  - wait() legt Thread schlafen bestimmte Zeitspanne oder bis er wieder aufgeweckt wird
  - notify() weckt Thread wieder
  - weitere Methoden: getPriority(), isAlive(), interrupt(), getName()
- auch in C# Klasse „Thread“
  - eigene Klasse instantiiert Thread Klasse und weist Startmethode zu
    - `(new Thread(new ThreadStart(Startmethode))).start();`

Scheduling

Ziele

Fairness: jeder Prozess erhält garantierte Mindeszuteilung
Effizient: möglichst volle Auslastung der CPUs
Antwortzeit: soll minimiert werden
Verweilzeit: Wartezeit von Prozessen soll möglichst klein sein (Ankunft – Ende)
Durchsatz: Maximierung der Aufträge, die ein BS in einem Zeitintervall durchführt

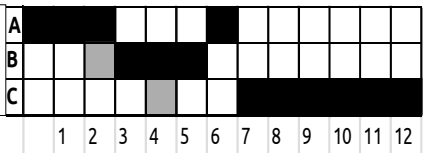
Preemptives vs. non-preemptives Scheduling

- Non-Preemptive Scheduling (auch Run-To-Completion-Verfahren)
  - Prozess wird nicht unterbrochen, bis er fertig ist
  - nicht geeignet für konkurrierende Benutzer im Dialogbetrieb
  - z. B. MS-DOS
- Preemptives Scheduling: rechnende Prozesse können verdrängt werden

Verhungern/Starvation

Prozesse erhalten die CPU nie, z.B. SJF

Prozess	A	B	C
Ankunfszeit	0	2	4
Rechenzeit	4	3	6



Algorithmen

- First-Come-First-Serve (FCFS): nach Reihenfolge des Eintreffens
- Round Robin (RR): Rundlauf-Verfahren, siehe Beispiel oben
  - FCFS in Verbindung mit Zeitscheibe, abhängig von Länge der Zeitscheibe
  - wichtig: Kontextwechsel kostet auch Zeit
  - meist mit Prioritäten
  - Länge des Quantums: ca. 10 bis 200 ms
- Shortest-Job-First (SJF): Prozess mit kürzester Bedienzeit als nachstes
- Shortest Remaining Time Next (SRTN): Prozess mit kürzester verbleibender Restrechenzeit als nächstes
- Lottery: zufällige Vergabe von CPU-Zeit
- Priority Scheduling (PS): Prozess mit höchster Priorität als nächstes

Echtzeit

- „echtzeitfähig“: garantierte Einhaltung von Zeitschranken, Vorhersagbarkeit, Anpassung an die Geschwindigkeit der Umgebung; echtzeitfähig, wenn Tasks garantiert immer Deadlines einhalten
- Echtzeit-BS vs. PC-BS

Echtzeit-BS	Universalsystem

unterstützt harte Echtzeitanforderungen(Garantien)	unterstützt nur "weiche" Echtzeitanforderungen
ist auf Worst-Case optimiert	Optimierung auf Unterstützung verschiedenster Anwendungsfälle, Reaktionszeit nicht im Vordergrund
"Dringende" Prozesse verdrängen weniger wichtige	Alle Prozesse werden (weitgehend) gleich (fair) behandelt
Vorhersagbarkeit wichtig	Durchsatz und Antwortzeitverhalten
Sofortige Bearbeitung anstehender Aufträge (Ein-/Ausgabe)	Blockweise Ein-/Ausgabe, Sammlung "gleicher Aufträge"

- **Modell**
  - Tasks: Prozess oder Thread; wird periodisch ausgeführt und hat bestimmte Priorität
  - Periode *T*: Zeiteinheiten bis sich Task wiederholt (vom letzten *Start* des Prozess)
  - Compution Time *C*: längste zu erwartende Berechnungszeit
  - Deadline *D*: Zeitpunkt bis zu dem die Berechnung jeweils erfolgt sein muss
  - Prioritäten: rechenbereite Task mit der höchsten Priorität als nächstes ausgewählt; Tasks mit hoher Priorität *unterbrechen* Tasks mit niedriger
  - Wartezeit: Warten auf Start und während Unterbrechungen
- Scheduling
  - statisch: Rate-Monotonic-Scheduling (RMS)
    (je kürzer Periode desto höher die Priorität)
  - dynamisch: Earliest Deadline First (EDF)
    (Task mit nächster absoluter Deadline hat höchste Priorität)
- Major Cycle: danach wiederholt sich Verhalten des Systems; kleinstes gemeinsames Vielfaches von den Perioden
- Critical Instant: alle Tasks wollen zur selben
- Zeit starten (worst-case); wird dieser ohne Deadline-Überschreitung überstanden, so werden immer alle Deadlines eingehalten è ich muss nur den Zeitraum vom CI bis zur max. Periode betrachten
- maximale Wartezeit: Wartezeit in der 1. Periode
- Kontextwechsel: PCB sichern, Register, …
- Bewertung: RMS nicht so aufwändig wie EDF; SJF ist am besten, wenn man nur Verweilzeit betrachtet; ist aber schwer zu realisieren, da die benötigte Prozessorzeit initial nicht bekannt ist

Synchronisation

Kritische Abschnitte: darf Prozess nur exklusiv bearbeiten
Anforderungen:

- keine 2 Prozesse dürfen gleichzeitig in einem sein
- keine Annahmen über die Abarbeitungsgeschwindigkeit und die Anzahl der Prozesse/Prozessoren
- kein Prozess außerhalb eines kritischen Abschnitts darf einen anderen Prozess blockieren
- kein ewiges Warten (fairness condition)

Race Condition: 2 oder mehr Prozesse nutzen ein gemeinsames Betriebsmittel und Endergebnisse der Bearbeitung sind von der zeitlichen Reihenfolge abhängig

Busy Waiting und Spin Locks:

- Prozess testet Synchronisationsvariable solange, bis Variable einen Wert hat, der den Zutritt erlaubt
- unwirtschaftlich, da es Verschwendung der CPU-Zeit bedeutet
  Aber: Spinlocks in BS oft anzutreffen (bei sehr kurzen Wartezeiten bei Multiprozessoren gut!)
- Spinlocks: Prozess A läuft solange in Endlos-Schleife bis B benötigte Daten liefert

Mutal Exclusion/gegenseitiger Ausschluss: Mutex-Verfahren verhindern, dass nebenläufige Prozesse bzw. Threads gleichzeitig oder zeitlich verschränkt gemeinsam genutzte Datenstrukturen unkoordiniert verändern

Semaphore

Initialisierungswert: meistens Anzahl der max. verfügbaren Ressourcen
Methoden: Up(unlock), down(lock)

Mutex: binäres Semaphore (ohne Zähler), leicht und effizient zu Implementieren, nur 2 Zustände (locked, unlocked)

Beispiel: zählendes Semaphor

drei Prozesse, die Ampel steuern; in allen Prozessen Zählervariabe, die Anzahl freier Plätze angibt, zur Wahrung der Daten-Konsistenz binäres Semaphor

Beobachterprozess (E) zählt Autos die einfahren	Beobachterprozess (A) zählt Autos die ausfahren	Beobachterprozess (B) prüft Zählstand und stellt Ampel entsprechend ein

`while(true) {`
`  wait(Lichtschranke);`
`  down(s);`
`  --n;`
`  up(s);`
`}`
	`while(true) {`
	`  wait(Lichtschranke);`
	`  down(s);`
	`  ++n;`
	`  up(s);`
	`}`
		`while(true) {`
		` sleep(1);`
		` down(s);`
		` if(n == 0) {`
		`  ampel(rot);`
		` } else {`
		`  ampel(grün);`
		` }`
		` up(s);`
		`}`

Atomare Operationen

Hardware-Unterstützung zur Synchronisation (alternativ: Interrupts ausschalten)

Test and Set Lock – TSL-Befehl: Parameter `Reg Adr`

```
Reg = *Adr
*Adr = 1
```

(2 Schritte als atomare Operation)

Exchange -Befehl XCHG: Parameter dest, src Inhalte von src und dest werden vertauscht

Außerdem: Swap (Austausch zweier Werte in einem Zyklus), Fetch And Add (Lesen und Inkrementieren eines Wertes in einem Zyklus)

Monitore

Synchronisationsvariablen (Conditions) und Schutz gemeinsam genutzter Daten durch Sperre

Struktur/Elemente

- Methoden
  - wait(): Monitor wird temporär verlassen (warten auf Betriebsmittel), versetzt Prozess in Wartezustand
  - signal(): Signal an wartenden Prozess, dass Monitor nun betreten werden kann, weckt wartenden Prozess wieder auf; ist Warteschlange leer, bleibt Signal ohne Wirkung
- Bedungungsvariablen: Schlossvariable bei der Prozesse warten
- Vorteile: regelt Programmiersprache, Programmierer muss sich um Mutal Exclusion nicht selbst kümmern
- Nachteil: Benstandteil der Programmiersprache

Beispiel: Java

- der Modifier `synchronized` dient der Festlegung kritischer Abschnitte: einzelne Codeblöcke, ganze Methoden mit allen Objekten
- Thread-safe: Klasse/Methode, die bedenkenlos in einer nebenläufigen Thread-Umgebung genutzt werden kann
  Problematisch: Klassenvariablen, globale Variablen
- `public synchronized void method1(){ //geschützter Codebereich }`
- Zugriffsserialisierter Anweisungsblock: `XyObject object1 = new`
- `XyObject(...);synchronized (object1){ //geschützter Codebereich }`
- Anwendung auf ganze Methoden: alle benutzten Objekte werden zur Laufzeit vor Zugriffen gesperrt; Sperre wird gehalten, bis Methode abgearbeitet ist; Modifier `static`: ganze Klasse wird gesperrt

- synchronizd wird in der JVM über eine Monitorvariante mit einer Sperre und einer Condition-Variable implementiert
- für jedes Objekt, das mindestens eine `synchronized`-Methode hat, wird von der JVM ein eigener Monitor ergänzt
- Monitor realisiert Sperren + Warten
- Sperre wird aufgehoben, wenn Thread die Methode verlässt

„Klassische" Probleme

Dining Philosophers

```
enum State {Thinking, Hungry, Eating} state[philosopherCount];
const int philosopherCount = 5;
Semaphore mutex = 1; // wechselseitiger Ausschluss für krit. Abschn.
Semaphore s[philosopherCount]; // ein Sem. pro Philosoph (mit 0 init.)
int left(int i) { return (i + philosopherCount-1) % philosopherCount; }
int right(int i) { return (i + 1) % philosopherCount; }
void philosopher(int i) {
	for(;;) {
		think();
		takeForks(i); // nimm 2 Gabeln oder blockiere
		eat();
		putForks(i); // lege Gabeln zurück und wecke ggf. Nachbarn
	}
}
void takeForks(int i) {
	Down(mutex); // Eintritt in kritischen Abschnitt
	state[i] = Hungry;
	test(i); // versuche 2 Gabeln zu bekommen
	Up(mutex); // verlasse kritischen Abschnitt
	Down(s[i]); // blockiere, falls Gabeln nicht bekommen
}
void putForks(int i) {
	Down(mutex); // Eintritt in kritischen Abschnitt
	state[i] = Thinking;
	test(left(i)); // kann/will linker Nachbar jetzt essen
	test(right(i)); // kann/will rechter Nachbar jetzt essen
	Up(mutex); // verlasse kritischen Abschnitt
}
void test(int i) {
	if(state[i] == Hungry && state[left(i)] != Eating
		&& state[right(i)] != Eating) {
		state[i] = Eating;
		Up(s[i]);
	}
}
```

Producer/consumer

- ein / mehrere Erzeugerprozesse (producer) produzieren
- ein / mehrere Verbraucherprozesse (consumer) konsumieren
- endlich große Pufferbereiche zwischen den Prozessen
- Flusskontrolle erforderlich! è Erzeuger legt sich schlafen, wenn Puffer voll ist;
- Verbraucher weckt ihn, wenn wieder Platz ist; Verbraucher legt sich schlafen, wenn
- Puffer leer ist; Erzeuger weckt ihn, wenn wieder was drinnen ist
- Lösung mit 3 Semaphoren: Mutex für den gegenseitigen Ausschluss beim
- Pufferzugriff; frei und belegt zur Synchronisation

Eigene Semaphor-Implementierung

Deadlocks

Bedingungen

Mutal Exclusion: jedes beteiligte Betriebsmittel ist entweder exklusiv belegt oder frei

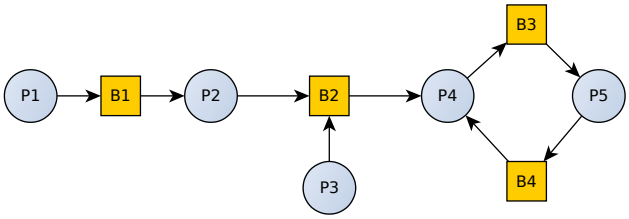
Hold and Wait: Prozesse belegen bereits exklusiv Betriebsmittel (mind. eins) und fordern noch weitere an

No Preemption: kein Entzug eines Betriebsmittels möglich, Prozesse müssen sie selbst wieder zurückgeben

Circular Waiting: zwei oder mehr Prozesse müssen in einer geschlossenen Kette auf Betriebsmittel warten, die der nächste reserviert hat

Betriebsmittelbelegungsgraph

Rechteck: Betriebsmittel/Resource
Kreis: Prozess



Eingehend: hat Ressource schon (gelockt)
Ausgehend: wartet auf Resource
Zyklus → Deadlock

Strategien

- Ignorieren („Vogel Strauß“ Strategie): sinnvoll, wenn Deadlocks nur sehr selten auftreten und Kosten für Vermeidung sehr hoch
- Erkennen und Beheben: grundsätzlich zugelassen, werden zur Laufzeit erkannt (Betriebsmittelbelegungsgraphen); Beheben durch: Unterbrechung (entziehen einer Ressource), Rollback (starte Prozess neu ab letztem Checkpoint), Prozessabbruch, Transaktionsabbruch
- Verhindern: Banker-Algorithmus (siehe Beispiel)
- Vermeiden: Mutal exclusion (unvermeidbar), Hold-and-wait (alle benötigten Ressourcen auf einen Schlag), No preemption (keine Option), Circular Wait (Ressourcen dürfen nur in vorher definierter Reihenfolge angefordert werden), Priority Ceiling Protocol (sobald ein Task Ressource belegt, bekommt sie Ceiling Priorität der Ressource → Unterlaufen der Circular Wait Bedingung)

Banker Algorithmus mit Beispiel

Zustand ist sicher, wenn es eine Schedulingreihenfolge gibt, die nicht zum Deadlock führt bzw. wenn jeder Prozess maximale Betriebsmittel ausnutzt

	Belegt				Maximal			
	B1	**B2**	**B3**	**B4**	**B1**	**B2**	**B3**	**B4**
A	3	0	1	1	4	1	1	1
B	0	1	0	0	0	2	1	2
C	1	1	1	0	4	2	1	0
D	1	1	0	1	1	1	1	1
E	0	0	0	0	2	1	1	0
frei	1	1	2	0	6	4	4	2

1. D kann beendet werden: 0 0 1 0 — dann frei: 2 2 2 1
2. E kann beendet werden: 2 1 1 0 — dann frei: 2 2 2 1
3. A kann beendet werden: 1 1 0 0 — dann frei: 5 2 3 2
4. C kann beendet werden: 3 1 0 0 — dann frei: 6 3 4 2

B kann beendet werden: …

→ Zustand ist sicher (es wären auch noch andere Reihenfolgen möglich)

Nachteile: Betriebsmittel-Bedarf nicht vorhersagbar, kostet viel Rechenzeit und Speicherplatz, garantiert nur dass jeder seine Betriebsmittel in endlicher Zeit bekommt

Bedingung für Deadlocks bei geg. Anzahl von Prozessen und Ressourcen

System hat p Prozesse und r Ressourcen, von denen jeder Prozess maximal m benötigt
→ Bedingung, dass Deadlocks unmöglich sind: $r > p(m-1)+1$

Kommunikation

Pipes: spezieller unidirektionaler Mechanismus, auch bidirektional über mehrere Pipes

- Erzeugen: pipe(), popen(), CreatePipe()
- Schließen: close(), pclose(), closeHandle()
- Elternprozess erzeugt Pipe und vererbt sich an Kindprozess
- blockierend (Pipe voll → Sendeprozess blockiert, Pipe leer → Leseprozess blockiert)

Speicherverwaltung

Speicherhierarchie

Caching

Ersetzungsstrategien:

- Least Recently Used (LRU): Austausch des am längsten nicht mehr referierten Blocks → Alterungszähler bei Zugriff auf 0 und alle andern +1 → Cache mit höchstem Zählerstand wird ausgelagert
- Least Frequently Accessed (LFA): Austausch des am wenigsten benutzten Blocks → Referenzzähler bei Zugriff um 1 erhöhen und nach Austausch bzw. Zeitintervall auf 0
- Least Recently Loaded (LRL): Austausch des am längsten geladenen Cache-Blocks → Feshalten und späterer Vergleich des Ladezeitpunktes (FIFO)
- Zufallsauswahl

Schreiboperationen

- Durchschreiben (write through): jede Schreiboperation auf Cache wird sofort auf
- unterliegenden Speicherebene durchgeführt → Gewährleistet größtmögliche Übereinstimmung, aber zeitaufwendig
- Zurückschreiben (write back): zunächst nur auf Cache, erst beim Austausch auch
- auf langsamere Ebene→ zeitlich günstiger, Dateninkosistenz bei Mehrprozessorsystemen mit mehreren Caches
- Schreiben auf Anforderung (write on demand): Schreiboperationen auf langsamere Ebene nur auf besondere Anforderung
  → günstig für Cache-Inhalte mit begrenzter Lebensdauer (Unterprogrammdaten)
- Trefferrate: abhängig von Cache-Größe, Ersetzungsstrategie, Programm- und Dateiorganisation

Memory Manager/MMU

verwaltet die freien und belegten Speicherbereiche

Lokalitätsprinzip

örtlich: nächster Daten/Code-Zugriff ist mit hoher Wahrscheinlichkeit in der Nähe der vorherigen Zugriffe → benachbarte Daten beim Zugriff auch gleich laden

zeitlich: Daten/Code-Bereiche, die gerade benutzt werden, werden mit hoher Wahrscheinlichkeit gleich wieder benötigt → sollten für nächsten Zugriff bereitgehalten werden

Strategien

Feste Partitionen

- Aufteilung des Speichers in feste Teile
- Multiprogramming und Verbesserung der CPU-Auslastung möglich
- Job wird in eine Queue eingetragen (für jede Partition eine Queue oder eine globale Queue)

Speicherverwaltung bei Multiprogramming mit Swapping

- Grundgedanke: Timesharing
  - Es passen nicht immer alle Prozesse in den Hauptspeicher
  - Prozess wird im Gesamten geladen
  - Prozess wird nach einer gewissen Zeit wieder auf einen Sekundärspeicher ausgelagert
  - Entstehende Löcher können durch Kombination benachbarter Speicherelemente eliminiert werden, aber aufwändig!
- Hauptunterschied zu festen Partitionen
  - Anzahl, Speicherplatz und Größe des für einen Prozess verwendeten Speicherbereichs variieren dynamisch
  - Prozess wir immer dahin geladen, wo gerade ausreichend Platz ist

Buddy Algorithmus (Fragmentierung vermeiden)

Speichervergabe: suche nach kleinstem geeigneten Bereich, Halbierung des gefundenen Bereichs solange bis gewünschter Bereich gerade noch in Teilbereich passt

Speicherfreigabe: zurückgegebenen Bereich mit allen freien Nachbarbereichen verbinden und zu einem Bereich machen

Fragmentierung

Externe Fragmentierung: entsteht durch dynamische Zuteilung von Speicherbereichen

Interne Fragmentierung: im Durchschnitt bleibt immer die Hälfte der letzten Seiten eines Segments leer → bei n Segmenten und Seitengröße p: $n * p / 2$

Virtueller Speicher

- Speichergröße eines Programms inklusive Daten und Stack darf den vorhandenen physikalischen Hauptspeicher überschreiten
- Prozess kann auch ablaufen, wenn er nur teilweise im Hauptspeicher ist
- Programmierer soll sich am besten nur mit einem linearen Adressraum befassen müssen
- Betriebssystem hält gerade benutzten Teile im Hauptspeicher und den Rest auf einer Festplatte
- primäre Nutzung in Multiprogramming-Systemen
- Begriffe
  - Virtueller Adressraum (das was Prozess sieht) und Realer Adressraum (RAM)
  - Seiten (Pages) und Seitenrahmen (Frames)
  - **Pagin Area (Schattenspeicher)**: für Hauptspeicher wird Schattenspeicher in einem speziellen Plattenbereich reserviert
  - Mapping: Page → Frame
- Strategien zur Verwaltung von virtuellen Speicher
  - Ablaufstrategie (Fetch Policy): Strategie zur Auswahl zu verdrängender Seiten
    - Demand Paging: Seiten werden erst bei Bedarf in Hauptspeicher geladen
    - Prepaging: Seiten, die noch nicht angefordert wurden schon vorher in Hauptspeicher laden (aktuell benötigte Seitenmenge: Working Set)
      Annahmen: Set ändert sich nur langsam; benötigte Seiten befinden sich in Nähe der gerade ersetzten
  - Speicherzuteilungsstragegie (Placement Policy)
    - Ziel: Vermeindung von Fragmentierung
    - Belegung des HS in Tabellen verwaltet (z. B. Bit Map → freie Belegung erkennt man an nebeneinanderliegenden Nullen)
    - sequentielle Suche: erster geeigneter Bereich wird vergeben (First-Fit)
    - optimale Suche: passendster Bereich, um Fragmentierung zu vermeiden
    - Buddy-Technik (ausführlicher bereits weiter oben beschrieben)
  - Austauschstrategie (Replacement Policy)
  - Aufräumstrategie (Cleaning Policy)
    - legt Zeitpunkt fest, wann eine modifizierte Seite auf Pagin-Area geschrieben wird
    - Demand Cleaning: bei Bedarf; Seite lang im HS; dafür Verzögerung bei Seitenwechsel
    - Precleaning: präventieves Zurückschreiben, wenn Zeit ist; Frames i. d. R. verfügbar
    - Page Buffering: Listen verwalten; modified List: wird zwischengepuffert; unmodified List: für Entladen freigegeben
- **Paging**: Umlagerung zwischen Hauptspeicher und Platte
  - Jeder Prozess darf alle Adressen verwenden, die aufgrund der Hardwarearchitektur des Rechners möglich sind (unabhängig von der realen Größe des Hauptspeichers)
  - Bei Systemen mit 32-Bit-Adressen kann jeder Prozess einen Adressraum von 4 GB verwenden
  - **Memory Management Unit (MMU)**: Hardwareunterstützung für Mapping
    CPU sendet virtuelle Adressen an MMU, MMU sendet reale Adressen an HS
  - Speicherbild eines Prozesses besteht aus Speicherseiten
    - Speicherseite ist Segment einer vorgegebenen Größe (z. B. 4 KB)
    - Verwaltung mittels **Seitentabellen**
      P-Bit: Seite im HS, R-Bit: Zugriff erfolgt, M-Bit: verändernder Zugriff erfolgt
    - nur wirklich benötigte Speicherseiten müssen im Arbeitsspeicher geladen sein, während Prozess läuft
    - Größe der Seiten meist gering, Anzahl beliebig groß
  - Virtuelle Adresse wird in virtuelle Seitennummer und einen Offset geteilt
  - Virtuelle Seitennummer ist Index auf Seitentabelle
  - Über Index wird zugehöriger Eintrag in Seitentabelle gefunden
  - Im Eintrag steht Frame-Nummer, falls Seite Frame zugeordnet ist
  - **Page fault***:* Befindet sich eine angesprochene Adresse nicht im Hauptspeicher, verursacht

MMU bei CPU einen Trap (*page fault*)
  - Mehrstufige Adressumsetzung: nicht immer alle Seitentabellen gleichzeitig im HS
  - Optimierung durch Adressumsetzpuffer/Translation Lookaside Buffer (TLB)
    - Cache der einige Einträge der Seitentabelle enthält, also Zuordnung von virtuellen auf reale Adressen (für aktuell am häufigsten benötigte Adressen)
    - bei Adressumsetzung wird zuerst in TLB geschaut
      (bei Hit kein Zugriff auf Seitentabelle notwendig)
    - beträchtliche Leistungsoptimierung möglich
    - TLB ist Bestandteil der MMU und enthält: virtuelle Seitennummer, Verweis auf Seitenrahmen im Hauptspeicher, Tag zur Adressraum-Identifikation (z. B. Prozessidentifikation), PID → Tagged TLB
  - Optimierung durch invertierte Seitentabellen
    - nur eine Tabelle, in der man Adressen auf virtuelle abbildet, also *invertiertes* vorgehen → invertierte Sicht
    - Vorteil: wesentlich weniger Tabelleneinträge: nur noch so viele wie man Seitenrahmen im Hauptspeicher hat
    - Nachteil: in Seitentabelle keine Ordnung nach virtuellen Adressen
      → Suche etwas aufwändiger, da nicht über Seitentabellenindex positioniert werden kann
    - Kombination mit TLB üblich
- Strategien zur Speicherumsetzung
  - optimal (Belady): am wenigsten Ersetzungen sind erforderlich, wenn man Seiten zu Verdrängung auswählt, die am spätesten in Zukunft benutzt werden (schwer zu realisieren, nur als Referenz)
  - *FIFO*: älteste Seite wird ersetzt
    - *Second Chance*: auch R-Bit wird inspiziert (Aging); ist älteste Seite schon benutzt, wird sie nicht ausgelagert, sondern ans Ende gehängt und R-Bit gelöscht
    - *Clock Page*: Seiten werden in zirkulierender Liste verwaltet (wie Uhr); bei Page fault wird immer Seite untersucht, auf die gerade der „Uhrzeiger" verweist, der Seitentabelleneintrag wird nicht umgehängt
      R-Bit 0 → Auslagerung; R-Bit 1 → R-Bit 0 setzen und Zeiger weiter verschieben
  - Not Recently Used (NRU): Seiten, die in letzter Zeit nicht genutzt wurden
    - R-Bit wird periodisch zurückgesetzt, M-Bit nicht
    - 4-Klassen: R = 0, M = 0 → Seiten werden als erstes ausgelagert
      R = 0, M = 1 → verändert im vorhergehenden Intervall
      R = 1, M = 0 → nur lesender Zugriff im aktuellen Intervall
      R = 1, M = 1 → Seiten werden als letztes ausgelagert
    - Seitentabelleneintrag: nur R- und M-Bit notwendig
  - *Least Recently Used* (LRU): Seite wird ersetzt, deren letzte Nutzung zeitlich am weitesten zurückliegt, quantitative Zeitmessung notwendig
  - *Not Frequently Used* (NFU): Seiten ersetzen, die in einem Zeitintervall selten genutzt wurden (Zugriffszähler); bei Page fault wird Seite mit kleinsten Wert im Zähler zur Ersetzung ausgewählt
  - Aging
    - Zählerstände: „n x n"-Matrix (n=Seitenanzahl); Bei Zugriff auf Seite k: in Zeile k auf 1 und in Spalte k auf 0
    - alle um ein Bit nach rechts shiften; R-Bit von links einschieben
  - *WS Clock*
    - bei Page Fault:
      R-Bit = 1: Seite wurde benutzt, setze R-Bit=0, weiter zur nächsten Seite
      R-Bit = 0: Alter > t & M-Bit=0 → überschreiben; Alter > t & M-Bit=1 → Seite auf Festplatte schreiben und Zeiger vorrücken
    - wenn Uhrzeiger wieder am Anfang:
      mind. 1 Seite gespeichert → Zeiger läuft weiter bis M-Bit=0 gefunden
      keine Seite gespeichert è irgendeine (saubere) Seite

- Lokale vs. globale Ersetzung, Page Fault Frequency (PFF)
  - Lokal: jeder Prozess hat festen Speicherbereich (feste Anzahl Seitenrahmen), wächst Arbeitsbereich darüber hinaus viele Page Faults, ist Arbeitsbereich deutlich kleiner viel Speicher verschwendet
  - Global: BS entscheidet dynamisch, wie viel Speicher einem Prozess zugeteilt wird (PFF Algorithmus): Seitenfehlerrate zu groß (Linie A) → mehr Seitenrahmen zuteilen Seitenfehlerrate zu klein (Linie B) → Seitenrahmen entziehen

  - *copy-on-write* (z. B. fork()): auch Seitentabelle wird kopiert, dh. alle Einträge verweisen für beide Prozesse auf dieselben Seitenrahmen; erst wenn Prozess auf Seite schreibt, wird diese kopiert und Seitentabelle entsprechend aktualisiert
- Bewertung zum virtuellen Speicher
  - relative aufwändig (viele umfangreiche Tabellen benötigt, Teil der Festplatte als Paging-Area, Überwachung ob Seiten hauptspeicherresident bleiben oder auszulagernd sind)
  - Prozesse müssen nicht komplett speicherresident sein, um ablaufen zu können
  - Lineare Speicheradressierung, keine Fragmentierung aus Programmierer-Sicht
  - beim Prozesswechsel behält ein Prozess seine hauptspeicherresidenten Seiten (verliert sie erst, wenn sie von Verwaltung des realen Speichers verdrängt werden)
  - Anwenderprogramme können vollen virtuellen Adressraum nutzen, wenn genügend Festplattenspeicher vorhanden ist
  - tatsächlich zugewiesene reale Speicher ändert sich dynamisch
  - Speicherschutzmechanismen sind einfach zu realisieren
    → trotz Overhead: heute am meisten verwendetes Verfahren

Beispiel zur Adressumsetzung (MMU)

Virtuelle Adresse 22 222 in physikalische Adresse umrechnen

1. Adresse durch Seitengröße, hier 22 222 / 2048 = 10 Rest 1742
2. → Offset ist 1742 und Nummer der Seite 10 → Nummer des Seitenrahmens/Kachel, hier 2
3. alternativ Berechnung des Offsets: 22 222 – 10 * 2048 = 1742
4. physikalische Adresse: Startadresse der Kachel + Offset, hier 4096 + 1742 = 5838

Andersherum: physikalische Adresse 11 111 in virtuelle Adresse

1. 11 111 / 2048 = 5 Rest 871
2. → Offset ist 871 und Seitenrahmen/Kachel 5 → Seite, hier 9
3. alternativ Berechnung des Offsets: 11 111 – 5 * 2048 = 871
4. virtuelle Adresse: 9 * 2048 + 871 = 19 303

Seite und Seitenrahmen für physikalische Adresse 17363, die auf virtuelle Adresse 25555 abgebildet wird

25555 / 2048 = 12 Rest 979 → Seite 12 17363 / 2048 = 8 Rest 979 → Seitenrahmen 8

Beispiel: Format einer Virtuellen Adresse, größe der Seitenrahmen, Second-Level-Tabellen

geg.: virtueller Adressraum mit 32-Bit langen virtuellen Adressen; eine Adresse enthält jeweils 10 Bit für Index in Top-Level-Seitentabelle und 10 Bit für Index in Second-Level-Seitentabelle

Größe des Offsets: von 32 Bit bleiben 32 – 10 – 10 = 12 Bit
Größe der Seitenrahmen im HS: $2^{\text{Größe des Offsets}} = 2^{12} = 4096 = 4$ KiB

Anzahl der Second-Level Seitentabellen max. je Prozess:

- jeder Eintrag in Top-Level-Seitentabelle verweist auf Second-Level-Seitentabelle
- $2^{10} = 1024$ Einträge → 1024 Second-Level-Seitentabellen (die auch 1024 Einträge haben)

gesamter virtueller Adressraum pro Prozess: $2^{32} = 4$ GiB

Geräteverwaltung

Aufgaben: Kommandos an Geräte senden, Daten empfangen und weiterleiten, Interrupts behandeln, Fehlerbehandlung, Verwaltung von externen Geräten zur Ein-/Ausgabe, leicht zu bedienende Schnittstelle zwischen Geräten und Rest des Systems

Block vs. Zeichenorientierte Geräte

blockorientierte Geräte: speichern Daten in Blöcken fester Größe, z. B. Festplatten

zeichenorientierte Geräte: erzeugen bzw. lesen Zeichenströme, nicht adressierbar, z. B. Maus,

Tastatur
sonstige: z. B. Uhren

Controller

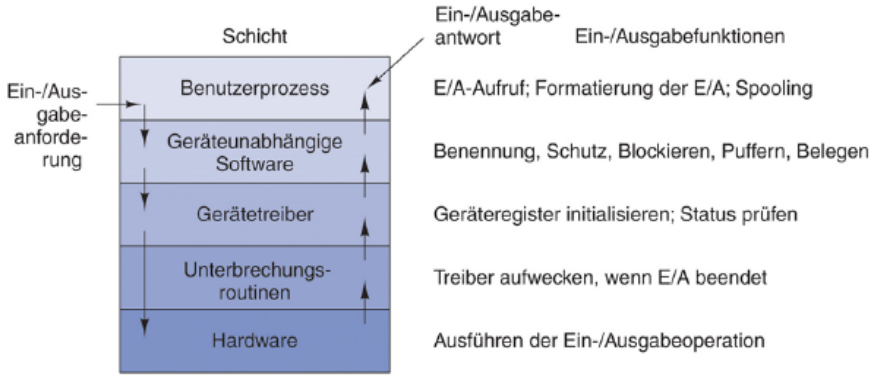
- Jedes Gerät besteht aus 2 Komponenten: mechanische Komponente, elektronische Komponente – Controller (Steuereinheit, Adapter)
- ein Controller kann auch mehrere Geräte verwalten
- Standard-Schnittstellen: IDE, SATA, SCSI, USB, FireWire, …
- Aufgaben: seriellen Bit-Stream in Byte-Blöcke konvertieren, Fehlerkorrekturen, Daten in den Arbeitsspeicher kopieren
- Schnittstelle eines Controllers ist für (fast) jedes Gerät anders
- Controller besitzt Menge von Kontrollregistern zur Ansteuerung
  - Schreiben in ein Register: Befehle erteilen, Aktionen auslösen
  - Lesen eines Registers: Statusinformation
- Außerdem: Datenpuffer, die gelesen/geschrieben werden können
- Zugriff auf die Kontrollregister kann auf zwei Arten erfolgen
  - Register liegen in einem eigenen, getrennten Adressraum
  - Register werden in den Arbeitsspeicher eingeblendet: Memory-Mapped I/O

Memory Mapped I/O

- Zugriff auf getrennten Bereich über spezielle Assembler-Befehle
- Kontrollregister liegen üblicherweise am Rand des Adressraums
- Adressen werden für jedes Gerät eindeutig vergeben
- Vorteile: kann in C/C++ programmiert werden, da keine speziellen Assemblerbefehle nötig sind, einfache Zugriffskontrolle
- Nachteile: Caching muss für betroffenen Seitenrahmen ausgeschaltet werden, bei jedem Speicherzugriff muss unterschieden werden, ob es sich um ein Kontrollregister handelt

Direct Memory Access (DMA)

- entlastet Prozessor
- DMA-Controller meist langsamer als Prozessor → DMA lohnt sich nur, wenn Prozessor tatsächlich etwas anderes zu tun hat



Treiber

- gerätespezifische Programmteile
- Aufgaben: Ansteuerung des Controllers (via Kontrollregister)
  - Initialisierung und Bekanntgabe des Geräts
  - Datenübertragung vom und zum Gerät
  - logisches Programmiermodell auf gerätespezifische Anforderungen überprüfen
  - Pufferung von Daten
  - Interruptbearbeitung
  - Koordination der nebenläufigen Zugriffe
- in eigenen Modulen, meist durch Gerätehersteller, realisiert
- sind aus Sicht des Betriebssystems „Fremd-SW“
- da im Betriebssystem integriert, kann Treiber gesamtes System zum Absturz bringen → Hauptursache für Blue Screens
- Integration von Treibern
  - statisch, dh. beim Kompilieren des Kerns werden alle Treiber eingebunden (bei eingebetteten Systeme, sonst eher selten)
  - dynamisch, d. h. Treiber werden ins laufende System nachgeladen (heute üblich)

Festplatten

- Jeder *Zylinder* enthält so viele *Spuren*, wie Köpfe vorhanden sind.

- Jede Spur ist in *Sektoren* eingeteilt.

Formatierung – Low Level

Konzentrische Spuren mit Sektoren anlegen; kurze Lücken zwischen den Sektoren

- Interleaving: Sektoren verschachtelt speichern, da Lesen eines Sektors, Fehlerkorrektur und Speichern im Puffer Zeit benötigen und nächster Sektor währenddessen nicht gelesen werden kann
- Spurversatz: da Spurwechsel gewisse Zeit dauert und Sektor 0 der nächsten Spur (ohne Versatz) nicht nahtlos gelesen werden könnte
- Präambel: Bitmuster zur Erkennung des Sektorbeginns, Zylinder- und Sektornummer
- Error Correction Code (ECC): Fehlerkorrektur
- Daten
- Reservesektoren: ersetzen ggf. defekte Sektoren
- Low-Level-Formatierung reduziert die verfügbare Plattenkapazität

Formatierung – High Level

Legt für eine Partition ein Dateisystem an. Je nach Art des Dateisystems, z. B.: Boot-Block, Freibereichsliste/Bitmap, Wurzelverzeichnis, FAT/MFT/I-Nodes, …

Steuerung des Plattenarms

Benötigte Zeit zum Lesen/Schreiben eines Plattenblocks:

- *Kopfpositionierungszeit*: Zeit, bis der Arm über der geforderten Spur steht
- *Rotationsverzögerung*: Zeit, bis geforderter Sektor unter Kopf steht
- Dauer der Datenübertragung

Kopfpositionierung dauert am längsten.

Strategien zur Optimierung der Reihenfolge der Zylinder-Zugriffe, mit Beispiel

Beispiel: Anzahl der Spurwechsel bei folgenden Anfragen: 11, 2, 38, 19 (ausgehend von 11)

- gerecht, aber nicht effizient: *First Come First Serve* (FCFS)
  also Anzahl der Spurchwechsel im Bsp. 11-11 + 11-2 + 38-2 + 38-19
- besser: *Shortest Seek First* (SSF), also Reihenfolge im Bsp.: 11, 19, 38, 2
- Aufzugalgorithmus: Bewegung immer nur in eine Richtung, dann umkehren, also Reihenfolge im Bsp. wenn Startrichtung abwärts: 11, 2, 19, 38 → …

Maximale Gesamtstrecke für eine beliebige Menge von Anfragen: 2 x Anzahl Zylinder

Linux

Gerätedatei: spezielle Datei unter Linux, die Gerät (z. B. Drucker) repräsentiert, durch Gerätezugriff erfolgt durch Lese-/Schreiboperationen auf Datei

Ladbare Module (u. a. auch Treiber) können zur Laufzeit in den Kern geladen werden
Schritte: benötigte Ressourcen reservieren (insbesondere Interruptnummern), Interruptvektoren aktualisieren, spezielle Dateistruktur enthält Zeiger auf Funktionen für open, close, read, …, Modul zum Initialisierung starten

Dateisysteme

Layout von Speichermedien

MBR – Partitionstabelle – Partition 1 (Boot-Block, Superblock, …) – Partition 2 – …

- Master Boot Record (MBR): auf Sektor 0, zum Booten des Rechnersystems
- Partitionstabelle
- Bootblock: wird beim Hochfahren gelesen und ausgeführt
- Superblock: enthält Verwaltungsinformationen zum Dateisystem (Anzahl der Blöcke, …)
- Fee Blocks: gibt freie Blöcke des Dateisystems an
- Rootverzeichnis: enthält Inhalt des Dateisystems (nach der Wurzel)
- I-nodes: Einträge im Inhaltsverzeichnis des Dateisystems

Implementierung von Dateien und Verzeichnissen

- Pseudodateien: Liste der defekten Blöcke eines Datenträgers
- Metadaten: Informationen über Datei selbst
- sequentieller Zugriff: Bytes werden nacheinander vom Anfang gelesen; Vor- oder Zurückspringen nicht möglich; für Magnetbänder
- wahlfreier Zugriff: Bytes können in beliebiger Reihenfolge gelesen werden;

unentbehrlich für Datenbanksysteme; Operationen: read (Lesen und Position erhöhen), seek (Position festlegen)
- Zusammenhängende Belegung: Problem der Fragmentierung
- Belegung durch verkettete Liste
  - sequentieller Zugriff unkompliziert
  - wahlfreier Zugriff extrem langsam
- verkettete Liste mit Tabelle im Arbeitsspeicher: File Allocation Table (FAT)
  - Vorteil: wahlfreier Zugriff schneller, verfolgen der Kette passiert im RAM
  - Nachteil: nicht praktikabel bei großen Platten
- Implementierung von Verzeichnissen
  - Verzeichniseintrag enthält Nummer des 1. Blocks oder I-Nodes
  - üblicherweise nicht sortiert (erst bei Ausgabe)
  - Dateiattribute: im Verzeichniseintrag oder im I-Node (Verzeichiseintrag enthält nur Verweis auf I-Node)
- Gemeinsam benutzte Dateien: Hard Link (Einträge verweisen auf gleichen I-Node), Symbolischer Link (enthält Pfad zur verlinkten Datei)
- Blockgröße
  - je mehr Daten mit einem Plattenzugriff geholt werden können, desto besser → Datenrate steigt mit Blockgröße
  - Bsp.: bei 4 KiB Dateien und Blockgröße 8 KiB werden 50 % Platz verschwendet → interne Fragmentierung von 50 %
  - übliche Blockgrößen: 1 KiB bis 4 KiB, bei mehreren TiB evtl. auch größere Werte
- **Konsistenz**
  - Read-Modify-Write: Blöcke lesen, verändern und zurück schreiben
    Was passiert, wenn System zwischenzeitlich abstürzt?
  - Dienstprogramme (z. B. fsck) prüfen und reparieren ggf. ein inkonsistentes Dateisystem: Prüfung von Blöcken oder von Dateien
    - Prüfen von Blöcken
      - 2 Tabellen mit Zählern für jeden Block
      - Tabelle 1 zählt, wie oft jeder Block in einer Datei vorkommt (benutzte Blöcke)
        - lese alle I-Nodes und daraus die Liste der Blöcke, die von der zugehörigen Datei belegt sind
      - Tabelle 2 zählt, wie oft jeder Block in der Freibereichsliste vorkommt (freie Blöcke)
        - verfolge die verkettete Liste bzw. durchsuche die Bitmap
      - Jeder Block sollte *genau einmal entweder in Tabelle 1 oder in Tabelle 2 auftauchen*
    - Block fehlt in beiden Tabellen: Lösung: Block in die Freibereichsliste einfügen
    - Block vorhanden in beiden Tabellen: Lösung: Block aus Freibereichsliste nehmen
    - doppelt vorhandener Block in Tabelle 1 (benutzte Blöcke)
      - falls eine der Dateien gelöscht wird → Block ist sowohl benutzt als auch frei
      - Lösung
        - kopiere Block und ordne Kopie einer der beiden Dateien zu
        - zusätzlich Benutzer informieren; eine der Dateien ist mit Sicherheit defekt
    - doppelt vorhandener Block in Tabelle 2 (freie Blöcke)
      - kann nicht passieren, wenn freie Blöcke in einer Bitmap verwaltet werden
      - Lösung: Freibereichsliste korrigieren
    - Prüfung auf Dateiebene
      - Zähle für jede Datei, wie oft sie in Verzeichnissen auftaucht (Hard Links)
      - Vergleiche das Ergebnis mit dem Link-Zähler im I-Node der Datei
      - Link-Zähler zu hoch
        - Nachdem die Datei in allen Verzeichnissen gelöscht wurde, wäre der Link-Zähler immer noch > 0, dh. die Datei existiert noch, taucht aber in keinem Verzeichnis mehr auf
        - Lösung: Link-Zähler korrigieren
      - Link-Zähler zu klein

        - Wird die Datei in einem Verzeichnis gelöscht, wird Link-Zähler = 0
          → I-Node und zugehörige Blöcke werden freigegeben, obwohl sie noch benutzt werden
        - Lösung: Link-Zähler korrigieren
      - Prüfung auf Dateiebene
        - Unsinnige Zugriffsrechte
        - Verzeichnisse mit ungewöhnlich vielen Einträgen
        - Dateien der Größe 0
        - ungewöhnlich große Dateien
  - Journaling-Dateisysteme (z. B. NTFS, ext3, ext4): Log-Eintrag wird vor Aktionen auf die Platte geschrieben und nachdem Aktionen ausgeführt wurden wieder gelöscht
    - falls System zwischendurch abstürzt, existiert Log-Eintrag anschließend noch, und Aktionen können wiederholt werden
  - Block-Cache (oder Puffer-Cache): einige MB groß, hält Menge von Blöcken im Speicher, Suche im Cache via Hashfunktion, da Cache relativ viele Blöcke enthält
    - wenn neuer Block wird geladen wird, muss anderer verdrängt werden (wird auf Platte geschrieben)
    - Aufwand für LRU-Liste hier vertretbar, da Cache-Referenzen relativ selten im Vergleich zu Seitenreferenzen und Aufwand für LRU-Liste klein im Vergleich zu Plattenzugriffen

CD-ROM-Dateisysteme

- ISO-9660
  - starke Einschränkungen (z. B. max. Verzeichnistiefe 8)
  - 3 Ebenen
    - restriktive Ebene: „8 + 3“ Dateinamen (8 Zeichen ohne Endung)
    - gelockerte Grenzen für Namenslängen (bis zu 32 Zeichen)
    - zusätzlich: Dateien müssen nicht zusammenhängend sein
      → Platzersparnis
  - Rock-Ridge-Erweiterungen: nutzt das *System Use* Feld für POSIX-Attribute, Gerätenummern, Symlinks, …
  - Joinlet-Erweiterungen: bis zu 64 Zeichen Dateinamen, Unicode, beliebige Verzeichnistiefe, Verzeichnisnamen mit Extension

MS-DOS-Dateisystem

- benutzt *File Allocation Table* (FAT)
- Versionen: FAT-{12,16,32}
- Verzeichniseintrag: Attribute (Schreibschutz, versteckt, Archiv, Systemdatei), Zeit, Datum (Jahr 2108 Problem), Größe (auf 2 GiB beschränkt)
- VFAT: lange Dateinamen (für Dateinamen, die „8 + 3“-Regel sprengen, wird primärer und mehrere sekundäre Verzeichniseinträge erzeugt
- Größe: 32-Bit-Zahl, daher max. Dateigröße 4 GB (aber beschränkt auf 2 GB)

UNIX

- V7-Dateisystem
  - 2 byte für I-Node Nr. → max. 65536 Dateien im Dateisystem
  - Verfolgung der Blöcke: bis zu 3-fach indirekt
- Linux
  - möglichst einfache Mechanismen, möglichst wenig Systemaufrufe
  - Konzepte: reiner Verzeichnisbaum, Links, mounting, Zugriffsrechte
  - Prozess-Dateisystem: /proc
    - für jeden laufenden Prozess ein Verzeichnis, z. B. /proc/345
    - Lesen/Schreiben möglich um Wete zu abzufragen bzw. zu setzen
  - ext2:
    - I-Nodes: Einträge im Inhaltsverzeichnis (für Beispiel siehe weiter unten)
    - jede Partition in Gruppen von Blöcken aufgeteilt
    - um Fragmentierung zu minimieren:

        - versuche immer, I-Nodes und zugehörige Datenblöcke in derselben Gruppe, d. h. nahe auf der Platte zu halten
        - Dateisystem möglichst über die gesamte Platte verteilt
        - Blöcke werden auch im Voraus für eine Datei reserviert
      - Superblock: Informationen über Dateisystem-Layout, Anzahl I-Nodes, Anzahl Plattenblöcke, …
      - Gruppendeskriptor: Position der Bitmaps, Anzahl freie Blöcke und I-Nodes in der Gruppe, Anzahl der Verzeichnisse in der Gruppe
      - Bitmaps: Verwaltung der freien I-Nodes und Blöcke, jede so groß wie ein Block (→ bei 1-KB Blöcken: je 8192 I-Nodes und 8129 Blöcke pro Guppe)
      - I-Nodes: jeder I-Node ist 128 bytes lang
      - Cache zur schnelleren Suche von kürzlich benutzten Verzeichnissen
    - ext3, ext4: erweitern ext2
      - Journaling Filesysteme: Journal ist Datei, die als Ringpuffer implementiert ist
      - ext3: verbesserter Umgang mit vielen Dateien (Verzeichnisse sind nicht linear, sondern als H-Baum angeordnet)
      - ext4: nochmals verbesserter Umgang mit vielen und großen Dateien, nanosekundengenaue Zeitstempel, verbesserte Zuverlässigkeit
    - btrfs: soll ext4 ablösen, Verzeichnisinhalte als B-Baum organisiert, Copy-On-Write, Snapshots, inkrementelles Backup, Datenkompression, Prüfsummen, integriertes RAID, max. Dateigröße: 2^{16}, max. Dateien: 2^{16}

Beispiel: I-Nodes

geg.: Blockgröße 1 KiB = 2^{10}, 256 Einträge pro Block

- Blockadresse: 1 KiB / 256 = 4 byte
- direkte Adressierung: 10 Blöcke
- 1-fache indirekte Adressierung: 256^1 2-fache indirekte Adressierung: 256^2 3-fache indirekte Adressierung: 256^3 …
- maximale Dateigröße: $10 + 256^1 + 256^2 + 256^3 + \ldots$

Network File System (NFS)

beliebig viele Clients/Server nutzen Dateisystem gemeinsam

NTFS

- hochkomplex, aber leistungsfähig (Dateikompression, Journaling, Verschlüsselung, alternative Datenströme)
- Plattenpartitionen heißen NTFS-Volumes
- Blockgröße: 0.5 … 64 KB (üblich: 4 KB)
- 64-Bit-Adressen für Blöcke
- wichtigste Datenstruktur: *Master File Table* (MFT)
  - jeder Eintrag in Tabelle ist 1 KB groß und beschreibt eine Datei oder ein Verzeichnis: Attribut, Liste von zugehörigen Blockadressen
  - freie MFT-Einträge werden mit einer Bitmap verwaltet
  - MFT ist selber eine Datei
  - erste 16 Einträge sind reserviert
  - Datenblöcke in Serien (aueinanderfolgende Blöcke) gespeichert → je mehr Fragmentierung, desto mehr Datensätze benötigt
- Kompression: Gruppen von 16 logischen Blöcken werden komprimiert gespeichert, wenn sie komprimiert tatsächlich 15 oder weniger Blöcke belegen, wahlfreier Zugriff wird schwieriger, da eine Serie ggf. erst dekomprimiert werden muss
- Alternative Datenströme: NTFS erlaubt mehrere Datenattribute für eine Datei, zusätzliche Datenattribute werden als alternative Datenströme (ADS) bezeichnet
  nützliche Anwendungen: getrennte Audio- und Video-Streams in einer Datei, Speichern eines Vorschaubildes zu einem Foto, Schadcode

Flash Speicher

- Bänke: einzelne Speicherchips in der Größe von 8 bis 2048 KB, typischerweise 128 KB

- Löschoperationen wirken immer auf gesamte Bank → jede Speicherzelle kann nur einmal beschrieben werden
- mit jeder Löschoperation wird Bank abgenutzt (hält ca. 100 000 bis 1 000 000 Löschoperation aus)
- Bank wird in Seiten/Sektoren von 512 … 8192 Byte eingeteilt, die immer komplett gelesen bzw. geschrieben werden müssen
- es muss auf gleichmäßige Abnutzung der Bänke geachtet werden
  → Löschzähler für jede Bank
  → für Schreiboperationen wird immer ein Sektor auf einer Bank mit niedrigstem Löschzähler gesucht
- wird ein Sektor geändert, wird er nie an dieselbe Stelle zurückgeschrieben
  - dazu müsste komplette Bank in Hauptspeicher gelesen und wieder gelöscht werden
  - stattdessen Copy-on-Write-Prinzip: freier Sektor wird gesucht, der ursprüngliche als obsolet markiert
  - sind alle Sektoren einer Bank obsolet, kann sie gelöscht werden
- Flash Translation Layer
  - Abbildung: Block → Sektor
  - Wear Levelling
  - Konsistenz, Recovery
- Abnutzungsausgleich
  - nur sehr selten sind alle Sektoren einer Bank obsolet
  - Bänke müssen daher hin und wieder frei gemacht und anschließend wieder gelöscht werden
  - zusätzlich werden „kalte“ und „heiße“ Speicherbereiche unterschieden
    - kalte Bereiche enthalten Daten, die sich sehr selten ändern
      → geringe Abnutzung
    - heiße Bereiche werden oft geändert → starke Abnutzung
- Speichern der Block-Zuordnung
  - jedesmal, wenn ein Block geschrieben wird, ändert sich seine Adresse → Tabelle muss aktualisiert werden
  - invertierte Tabelle:
    - Nummer des virtuellen Blocks steht im Header des physischen Sektors → beim Einschalten/Mounten alle Sektoren scannen
    - oder: ein reservierter Sektor pro Bank enthält Tabelle mit Zuordnung → beim Einschalten/Mounten alle Banken scannen
- Superblock, MBR
  - 2 oder mehr Banks reservieren
  - oder alles durchsuchen
- obsolete Blöcke
  FTL weiß erst mal nichts von gelöschten Dateien bzw. kennt Freiliste von Dateisystem nicht
  → Sektoren werden unnötig kopiert, obwohl sie einfach überschrieben werden könnten
  → TRIM-Kommando

Memory-Mapped-Dateien

Datei wird in Adressraum eines bzw. mehrerer Prozesse eingeblendet (POSIX-Funktion: mmap)

Virtualisierung

Nachbildung eines Hard- oder Software-Objekts durch ein ähnliches Objekt mit Hilfe einer Softwareschicht.

Ziele: Komplexität beherrschen, Anwendungsentwicklung vereinfachen durch einheitliche Schnittstelle

Virtuelle Maschine (VM)

- Nachbildung eines kompletten Rechners durch anderen Rechner
- Softwareschicht heißt Hypervisor oder Virtual Machine Monitor (VMM)
- Vorteile

    - mehrere unterschiedliche Betriebssysteme auf einem Rechner, bessere Nutzung leistungsfähiger Hardware durch mehrere virtuelle Server auf einem realen Rechner
    - problemloser Test unbekannter, nicht vertrauenswürdiger Programme
    - einfaches Sicherung/Wiederaufsetzen durch Erstellen von „Schnappschüssen“ (VM-Datei)
    - einfaches Kopieren (Klonen) einer VM
- Nachteile
    - VMs müssen sich real vorhandene Ressourcen auf einem Rechner teilen
    - Ausfall des realen Rechners → Ausfall aller VMs
    - Unterstützung von Spezial-HW schwierig
    - ca. 10 % Leistungsminderung

Emulation vs. Virtualisierung

Emulation: jeder einzelne Maschinenbefehl des nachgebildeten Rechners wird durch Software ausgeführt (langsam, Geschwindigkeitsverlust Faktor 5 bis 10)
mögliche Optimierung: Just-In-Time Compilation

Virtualisierung: Maschinenbefehle des nachgebildeten Rechners werden auf dem nachbildenden Rechner größtenteils nativ, also ohne SW-Eingriff, direkt durch dessen Hardware ausgeführt → wesentlich effizienter (ca. 10 % Leistungsminderung)
Voraussetzung: Prozessoren müssen ausreichend ähnlich sein

Voraussetzungen für Virtualisierung

- privilegierte Befehle: Aufruf von privilegierten Befehlen löst einen Sprung ins BS aus; es gibt privilegierte Befehle, die nur im Kernmodus ausgeführt werden dürfen → Trap ins BS
- kritische Befehle: privilegierte Befehle, die im Usermodus keinen Trap auslösen
- sensitive Befehle: zustandsverändernd (Zugriff auf I/O-Geräte, Adress- und Steuerregister) dürfen nur im Kernelmodus ausgeführt werden
- Mindestvoraussetzungen für virtualisierbare HW:
    - Unterscheidung zwischen Kernel- und Anwendungsmodus des Prozessors
    - Vorhandensein einer MMU zur Abschottung von Adressräumen der VMs
    - Kriterien nach Popek und Goldberg: Prozessor ist virtualisierbar, wenn alle privilegierten Maschinenbefehle eine Unterbrechung (Trap) erzeugen, wenn sie in einem unprivilegierten Prozessormodus ausgeführt werden; alle sensitiven Befehle sind auch privilegierte Befehle

Dynamische Übersetzung

- Ersetzung kritischer Befehle zur Laufzeit durch direkte Aufrufe an VMM
- Code des Gast-BS vor Abarbeitung in Puffer laden
- nach kritischen Befehlen durchsuchen
- durch direkte Aufrufe an VMM ersetzen
- beim nächsten Durchlaufen des Codes Abarbeitung der bereits veränderten Version im TC

Hypervisor bzw. Virtual Machine Monitor (VMM)

Aufgabe: Bereitstellen von isolierten Ablaufumgebungen in Form von „Duplikaten“ der realen Hardware → jede VM hat damit – scheinbar – Zugriff auf reale Hardware

Implementierung: fängt bestimmte Maschinenbefehle ab und ersetzt diese durch andere, ohne dass VM etwas davon merkt

Im Idealfall erkennt Gast-Betriebssystem nicht, dass es in einer virtuellen Maschine abläuft → es ist keine Modifikation des Gast-Betriebssystems nötig

Vorgehensweise: VMM läuft exklusiv im Systemmodus ab, Gast-Betriebssysteme befinden sich im Anwendungsmodus, Gast-Betriebssystem versucht, privilegierten Befehl auszuführen führt zur Aktivierung des VMM, VMM prüft Absicht des Gast-Systems, ersetzt gewünschte Aktion durch andere, auf virtuelle Maschine bezogene Aktion

Typ-1-Hypervisor

- direkt über Hardware als kleines MiniBS
- benötigt HW-Unterstützung für Virtualisierung im Prozessor
- VMM muss insbesondere alle Treiber zur Verfügung stellen

- Bsp.: XenServer von Citrix, Hyper-V von Microsoft, vSphere ESX von VMware

Typ-2-Hypervisor

- VMM läuft als Prozess unter einem HostBS
- sinnvoll, wenn HW keine Virtualisierungsunterstützung bietet
- kann Treiber des HostBS mitnutzen
- HostBS ist auch direkt zugänglich
- Bsp.: VMware, VirtualBox, VirtualPC

Paravirtualisierung

- zur Übersetzungszeit: Ersetzung kritischer Maschinenbefehle im Gast-BS durch
- direkte Aufrufe an VMM
- Vorteil: Unabhängigkeit von obengenannten Voraussetzungen, Durchsatzsteigerung
- Nachteil: Gast-BS muss im Quellcode zugänglich sein
- Vorwiegend im Linux-Bereich und in eingebetteten Systemen

Speicherverwaltung

- Die "reale" Speicher-Adresse, die ein Gast-Betriebssystem "sieht", ist tatsächlich (immer
- noch) virtuell.
- Abbildung: gastvirtuelle Adresse → gastphysische Adresse → physische Adresse (letztere
- ist Aufgabe des VMM)
- Schattentabellen = Software-Lösung
  - für jede VM eine Schattentabelle (im Adressraum des VMM)
  - übersetzt gastphysische Adresse in physische Adresse
  - VMM muss Änderungen des Gastbetriebssystems an seiner Seitentabelle überwachen und ggf. Schattentabelle aktualisieren
- Extended Page Tables: Hardware-Lösung (Teil der MMU)

Geräteverwaltung

- Gräte-Emulation
  - Standard bei fast allen Virtualisierungssystemen
  - VMM betreibt das Gerät via eigenem Treiber oder Treiber des HostBS
  - VMM bietet dem Gast-BS eine Schnittstelle zu einem „Standard“-Gerät
  - Gast-BS kann seine eigenen Treiber verwenden
  - evtl. schlechter Datendurchsatz, da jeder E/A Vorgang viele "Weltwechsel" nach sich zieht
- Direktzuweisung eines Geräts
  - Gerät wird einem Gast-BS exklusiv zugewiesen
  - Gast-BS interagiert via seinen eigenen Treiber direkt, also unter Umgehung des VMM, mit dem Gerät
  - effizienteste Möglichkeit
  - erfordert aber spezielle Hardware

Verschieben einer VM auf anderen Rechner im laufenden Betrieb

Hauptspeicherinhalt und Dateisystem werden von Quell-VM zur Ziel-VM kopiert, Teile des Speichers und Dateisystems werden während des Kopiervorgangs noch verändert (diese werden markiert und in einer weiteren oder ggf. mehreren Runden nachkopiert), Quell-VM wird angehalten und die letzten geänderten Teile nachkopiert, IP-/MAX-Adresse an Ziel-VM zuweisen, Datenverkehr umleiten, Ziel-VM starten, Quell-VM löschen