Funktionale und nichtfunktionale Eigenschaften

- Requirements: (nicht-)Funktionale Eigenschaften entstehen durch Erfüllung von (nicht-)funktionalen Anforderungen
- funktionale Eigenschaft: was ein Produkt tun soll
- nichtfunktionale Eigenschaft (NFE): wie ein Produkt dies tun soll
- andere Bezeichnungen NFE: Qualitäten, Quality of Service

Hardwarebasis

- Einst: Einprozessor-Systeme
- Heute: Mehrprozessor-/hochparallele Systeme
- neue Synchronisationsmechanismen erforderlich
- → unterschiedliche Hardware und deren Multiplexing

Betriebssystemarchitektur

- Einst: Monolithische und Makrokernel-Architekturen
- Heute: Mikrokernel(-basierte) Architekturen
- Exokernelbasierte Architekturen (Library-Betriebssysteme)
- Virtualisierungsarchitekturen
- Multikernel-Architekturen
- → unterschiedliche Architekturen

Ressourcenverwaltung

- Einst: Batch-Betriebssysteme, Stapelverarbeitung (FIFO)
- Heute: Echtzeitgarantien für Multimedia und Sicherheit
- echtzeitfähige Scheduler, Hauptspeicherverwaltung, Ereignismanagement, Umgang mit Überlast/Prioritätsumkehr ...
- → unterschiedliche Ressourcenverwaltung

Betriebssystemabstraktionen

- Reservierung von Ressourcen (→ eingebettete Systeme)
- Realisierung von QoS-Anforderungen (→ Multimediasysteme)
- Erhöhung der Ausfallsicherheit (→ verfügbarkeitskritisch)
- Schutz vor Angriffen und Missbrauch (→ sicherheitskritisch)
- flexiblen und modularen Anpassen des BS (→ hochadaptiv)
- → höchst diverse Abstraktionen von Hardware

Betriebssysteme als Softwareprodukte

- Betriebssystem: endliche Menge von Quellcode
- besitzen differenzierte Aufgaben → funktionale Eigenschaften
- Anforderungen an Nutzung und Pflege → Evolutionseigenschaften
- können für Betriebssysteme höchst speziell sein
- → spezielle Anforderungen an das Softwareprodukt BS

Grundlegende funktionale Eigenschaften von BS: Hardware-

Abstraktion Ablaufumgebung auf Basis der Hardware bereitstellen
Multiplexing Ablaufumgebung zeitlich/logisch getrennt einzelnen Anwendungen zuteilen
Schutz gemeinsame Ablaufumgebung gegen Fehler und Manipulation

Nichtfunktionale Eigenschaften (Auswahl) von Betriebssystemen:

- Laufzeiteigenschaften: zur Laufzeit eines Systems beobachtbar
  - Sparsamkeit und Effizienz
  - Robustheit, Verfügbarkeit
  - Sicherheit (Security)
  - Echtzeitfähigkeit, Adaptivität, Performanz
- Evolutionseigenschaften: charakterisieren (Weiter-) Entwicklung- und Betrieb eines Systems
  - Wartbarkeit, Portierbarkeit
  - Offenheit, Erweiterbarkeit

Sparsamkeit und Effizienz

Motivation

Sparsamkeit (Arbeitsdefinition): Die Eigenschaft eines Systems, seine Funktion mit minimalem Ressourcenverbrauch auszuüben → Effizienz bei Nutzung der Ressourcen
Effizienz: Der Grad, zu welchem ein System oder eine seiner Komponenten seine Funktion mit minimalem Ressourcenverbrauch ausübt. (IEEE)
Beispiele:

- mobile Geräte: Sparsamkeit mit Energie
- Sparsamkeit mit weiteren Ressourcen, z.B. Speicherplatz
- Betriebssystem (Kernel + User Space): geringer Speicherbedarf
- optimale Speicherverwaltung durch Betriebssystem zur Laufzeit
- Baugrößenoptimierung(Platinen-und Peripheriegerätegröße)
- Kostenoptimierung(kleine Caches, keine MMU, ...)
- massiv reduzierte HW-Schnittstellen (E/A-Geräte, Peripherie)

Mobile und eingebettete Systeme (kleine Auswahl)

- mobile Rechner-Endgeräte
- Weltraumfahrt und -erkundung
- Automobile
- verteilte Sensornetze (WSN)
- Chipkarten
- Multimedia-und Unterhaltungselektronik

Energieeffizienz

zeitweiliges Abschalten momentan nicht benötigter Ressourcen
Betriebssystemmechanismen

1. Dateisystem-E/A: energieeffizientes Festplatten-Prefetching
2. CPU-Scheduling: energieeffizientes Scheduling
3. Speicherverwaltung: Lokalitätsoptimierung
4. Netzwerk: energiebewusstes Routing
5. Verteiltes Rechnen: temperaturabhängige Lastverteilung

Energieeffiziente Dateizugriffe

HDD/Netzwerkgeräte/... sparen nur bei relativ langer Inaktivität Energie

- Aufgabe: kurze, intensive Zugriffsmuster → lange Inaktivität
- HDD-Geräten: Zustände mit absteigendem Energieverbrauch:
  1. Aktiv: einziger Arbeitszustand
  2. Idle: Platte rotiert, Elektronik teilweise abgeschaltet
  3. Standby: Rotation abgeschaltet
  4. Sleep: gesamte restliche Elektronik abgeschaltet
- ähnliche, noch stärker differenzierte Zustände bei DRAM
- durch geringe Verlängerungen des idle - Intervalls kann signifikant der Energieverbrauch reduziert werden

Prefetching-Mechanismus

- Prefetching („Speichervorgriff", vorausschauend) & Caching
  - Standard-Praxis bei moderner Datei-E/A
  - Voraussetzung: Vorwissen über benötigte Folge von zukünftigen Datenblockreferenzen
  - Ziel: Performanzverbesserung durch Durchsatzerhöhung und Latenzzeit-Verringerung
  - Idee: Vorziehen möglichst vieler E/A-Anforderungen an Festplatte + zeitlich gleichmäßige Verteilung verbleibender
  - Umsetzung: Caching dieser vorausschauend gelesenen Blöcke in ungenutzten PageCache
- Folge: Inaktivität überwiegend sehr kurz → Energieeffizienz ...?
- Zugriffs-/Festplattenoperationen
  - access(x) ... greife auf Inhalt von Festplattenblock x im PageCache zu
  - fetch(x) ... hole Block x nach einem access(x) von Festplatte
  - prefetch(x) ... hole Block x ohne access(x) von Festplatte
- Fetch-on-Demand-Strategie bisher (kein vorausschauendes Lesen)
- Traditionelles Prefetching
  - traditionelle Prefetching-Strategie: bestimmt
    - wann Block von der Platte holen (HW aktiv)
    - welcher Block zu holen ist
    - welcher Block zu ersetzen ist

  1. Optimales Prefetching: Jedes *prefetch* sollte den nächsten Block im Referenzstrom in den Cache bringen, der noch nicht dort ist
  2. Optimales Ersetzen: Bei jedem ersetzenden *prefetch* sollte der Block überschrieben werden, der am spätesten in der Zukunft wieder benötigt wird
  3. „Richte keinen Schaden an": Überschreibe niemals Block A um Block B zu holen, wenn A vor B benötigt wird
  4. Erste Möglichkeit: Führe nie ein ersetzendes *prefetch* aus, wenn dieses schon vorher hätte ausgeführt werden können
- Energieeffizientes Prefetching
  - versucht Länge der Disk-Idle-Intervalle zu maximieren

  1. Optimales Prefetching: Jedes *prefetch* sollte den nächsten Block im Referenzstrom in den Cache bringen, der noch nicht dort ist
  2. Optimales Ersetzen: Bei jedem ersetzenden *prefetch* sollte der Block überschrieben werden, der am spätesten in der Zukunft wieder benötigt wird
  3. „Richte keinen Schaden an": Überschreibe niemals Block A um Block B zu holen, wenn A vor B benötigt wird
  4. Maximiere Zugriffsfolgen: Führe immer dann nach einem *fetch/prefetch* ein weiteres *prefetch* aus, wenn Blöcke für eine Ersetzung geeignet sind
  5. Beachte Idle-Zeiten: Unterbrich nur dann eine Inaktivitätsperiode durch ein *prefetch*, falls dieses sofort ausgeführt werden muss, um Cache-Miss zu vermeiden

Allgemeine Schlussfolgerungen

1. Hardware-Spezifikation nutzen: Modi, in denen wenig Energie verbraucht wird
2. Entwicklung von Strategien, die langen Aufenthalt in energiesparenden Modi ermöglichen und dabei Leistungsparameter in vertretbarem Umfang reduzieren
3. Implementieren dieser Strategien in Betriebssystemmechanismen zur Ressourcenverwaltung

Energieeffizientes Prozessormanagement

Hardware-Gegebenheiten

- z.Zt. meistgenutzte Halbleitertechnologie für Prozessor-Hardware: CMOS (Complementary Metal Oxide Semiconductor)
- Komponenten für Energieverbrauch: $P = P_{switching} + P_{leakage} + ...$
  - $P_{switching}$: für Schaltvorgänge notwendige Leistung
  - $P_{leakage}$: Verlustleistung durch verschiedene Leckströme
  - ...: weitere Einflussgrößen (technologiespezifisch)

Hardwareseitige Maßnahmen

Schaltleistung: $P_{switching}$

- Energiebedarf kapazitiver Lade-u. Entladevorgänge während des Schaltens
- für momentane CMOS-Technologie i.A. dominanter Anteil am Energieverbrauch
- Einsparpotenzial: Verringerung von
  1. Versorgungsspannung (quadratische Abhängigkeit!)
  2. Taktfrequenz

- Folgen:
  1. längere Schaltvorgänge
  2. größere Latenzzwischen Schaltvorgängen
- Konsequenz: Energieeinsparung nur mit Qualitätseinbußen(direkt o. indirekt) möglich
  - Anpassung des Lastprofils (Zeit-Last-Kurve? Fristen kritisch?)
  - Beeinträchtigung der Nutzererfahrung(Reaktivität kritisch? Nutzungsprofil?)

Verlustleistung: $P_{leakage}$

- Energiebedarf baulich bedingter Leckströme
- Fortschreitende Hardware-Miniaturisierung → zunehmender Anteil von $P_{leakage}$ an P
- Beispielhafte Größenordnungen zum Einsparpotenzial: | Schaltkreismaße | Versorgungsspannung | $P_{leakage}/P$ | | ---------- | ---------- | ---------- | | 180 nm | 2,5 V | 0, | | 70 nm | 0,7 V | 0, | | 22 nm | 0,4 V | > 0,5 |
- Konsequenz: Leckströme kritisch für energiesparenden Hardwareentwurf

Regelspielraum: Nutzererfahrung

- Nutzererwartung: wichtigstes Kriterium zur (subjektiven) Bewertung von auf einem Rechner aktiven Anwendungen durch Nutzer → Nutzerwartung bestimmt Nutzererfahrung
- Typ einer Anwendung
  - entscheidet über jeweilige Nutzererwartung
    1. Hintergrundanwendung (z.B. Compiler); von Interesse: Gesamt-Bearbeitungsdauer, Durchsatz
    2. Echtzeitanwendung(z.B. Video-Player, MP3-Player); von Interesse: ,,flüssiges" Abspielen von Video oder Musik
    3. Interaktive Anwendung (z.B. Webbrowser); von Interesse: Reaktivität, d.h. keine (wahrnehmbare) Verzögerung zwischen Nutzer-Aktion und Rechner-Reaktion
  - Insbesondere kritisch: Echtzeitanwendungen, interaktive Anwendungen

Reaktivität

- Reaktion von Anwendungen
  - abhängig von sog. Reaktivität des Rechnersystems ≈ durchschnittliche Zeitdauer, mit der Reaktion eines Rechners auf eine (Benutzerinter-) Aktion erfolgt
- Reaktivität: von Reihe von Faktoren abhängig, z.B.:
  1. von **Hardware** an sich
  2. von **Energieversorgung** der Hardware (wichtig z.B. Spannungspegel an verschiedenen Stellen)
  3. von **Software-Gegebenheiten** (z.B. Prozess-Scheduling, Speichermanagement, Magnetplatten-E/A-Scheduling, Vorgänge im Fenstersystem, Arten des Ressourcen-Sharing usw.)

Zwischenfazit: Nutzererfahrung

- bietet Regelspielraum für Hardwareparameter (→ Schaltleistung) → Versorgungsspannung, Taktfrequenz
- Betriebssystemmechanismen zum energieeffizienten Prozessormanagement müssen mit Nutzererfahrung(jeweils erforderlicher Reaktivität) ausbalanciert werden (wie solche Mechanismen wirken: 2.2.3)
- Schnittstelle zu anderen NFE:
  - Echtzeitfähigkeit
  - Performanz
  - Usability
  - ...

Energieeffizientes Scheduling

- so weit besprochen: Beschränkung des durchschnittlichen Energieverbrauchs eines Prozessors
- offene Frage zum Ressourcenmultiplexing: Energieverbrauch eines Threads/Prozesses?
- Scheduling-Probleme beim Energiesparen:
  1. Fairness (der Energieverteilung)?
  2. Prioritätsumkehr?
- Beispiel: Round Robin (RR) mit Prioritäten (Hoch, Mittel, Niedrig)
- Problem 1: Unfaire Energieverteilung
  - Beschränkung des Energieverbrauchs (durch Qualitätseinbußen, schlimmstenfalls Ausfall)ab einem oberen Schwellwert E_{max}
  - Problem: energieintensive Threads behindern alle nachfolgenden Threads trotz gleicher Priorität → Fairnessmaß von RR (gleiche Zeitscheibenlänge T) untergraben
  - Problem 2: energieintensive Threads niedrigerer Priorität behindern später ankommende Threads höherer Priorität

Energiebewusstes RR: Fairness

- Begriffe:
  - E_i^{budget} ... Energiebudget von t_i
  - E_i^{limit} ... Energielimit von t_i
  - P_{limit} ... Leistungslimit: maximale Leistungsaufnahme [Energie/Zeit]
  - T ... resultierende Zeitscheibenlänge
- Strategie 1: faire Energieverteilung (einheitliche Energielimits)
  - $1\leq i\leq 4: E_i^{limit} = P_{limit}* T$
  - (Abweichungen = Wichtung der Prozesse → bedingte Fairness)

Energiebewusstes RR: Reaktivität

- faire bzw. gewichtete Aufteilung begrenzter Energie optimiert Energieeffizienz
- Problem: lange, wenig energieintensive Threads verzögern Antwort-und Wartezeiten kurzer, energieintensiver Threads
  - Lösung im Einzelfall: Wichtung per E_i^{limit}
  - globale Reaktivität (→ Nutzererfahrung bei interaktiven Systemen) ...?
- Strategie 2: maximale Reaktivität (→ klassisches RR)

Energiebewusstes RR: Reaktivität und Fairness

- Problem: sparsame Threads werden bestraft durch Verfallen des ungenutzten Energiebudgets
- Idee: Ansparen von Energiebudgets → mehrfache Ausführung eines Threads innerhalb einer Scheduling-Periode
- Strategie 3: Reaktivität, dann faire Energieverteilung

Implementierungsfragen

- Scheduling-Zeitpunkte?
  - welche Accounting-Operationen (Buchführung über Budget)?
  - wann Accounting-Operationen?
  - wann Verdrängung?
- Datenstrukturen?
  - ... im Scheduler → Warteschlange(n)?
  - ... im Prozessdeskriptor?
- Kosten ggü. klassischem RR? (durch Prioritäten...?)
- Pro:
  - Optimierung der Energieverteilung nach anwendungsspezifischen Schedulingzielen(→ Strategien)
  - Berücksichtigung von prozessspezifischen Energieverbrauchsmustern möglich:fördert Skalierbarkeit i.S.v. Lastadaptivität, indirekt auch Usability (→ Nutzererfahrung)
- Kontra:
  - zusätzliche sekundäre Kosten: Energiebedarf des Schedulers, Energiebedarf zusätzlicher Kontextwechsel, Implementierungskosten (Rechenzeit, Speicher)
  - Voraussetzung hardwareseitig: Monitoring des Energieverbrauchs (erforderliche/realisierbare Granularität...? sonst: Extrapolation?)
- **generelle Alternative:** energieintensive Prozesse verlangsamen → Regelung der CPU-Leistungsparameter (Versorgungsspannung) (auch komplementär zum Schedulingals Maßnahme nach Energielimit-Überschreitung)
- Beispiel: Synergie nichtfunktionaler Eigenschaften
  - Performanz nur möglich durch Parallelität → Multicore-Hardware
  - Multicore-Hardware nur möglich mit Lastausgleich und Lastverteilungauf mehrere CPUs
  - dies erfordert ebenfalls Verteilungsstrategien: ,,Energy-aware Scheduling" (Linux-Strategie zur Prozessorallokation -nicht zeitlichem Multiplexing!)

Systemglobale Energieeinsparungsmaßnahmen

- Traditionelle Betriebssysteme: Entwurf so, dass zu jedem Zeitpunkt Spitzen-Performanzangestrebt
- Beobachtungen:
  - viele Anwendungen benötigen keine Spitzen-Performanz
  - viele Hardware-Komponenten verbringen Zeit in Leerlaufsituationen bzw. in Situationen, wo keine Spitzen-Performanz erforderlich
- Konsequenz (besonders für mobile Systeme) :
  - Hardware mit Niedrigenergiezuständen(Prozessoren und Magnetplattenlaufwerke, aber auch DRAM, Netzwerkschnittstellen, Displays, ...)
  - somit kann Betriebssystem **Energie-Management** realisieren

Hardwaretechnologien

- DPM: Dynamic Power Management
  - versetzt leerlaufende/unbenutzte Hardware-Komponenten selektiv in Zustände mit niedrigem Energieverbrauch
  - Zustandsübergänge durch Power-Manager (in Hardware) gesteuert, dem bestimmte *DPM*- Strategie (Firmware) zugrunde liegt, um gutes Verhältnis zwischen Performanz/Reaktivität und Energieeinsparung zu erzielen
- DVS: Dynamic Voltage Scaling
  - effizientes Verfahren zur dynamischen Regulierungvon Taktfrequenz gemeinsammit Versorgungsspannung
  - Nutzung quadratischer Abhängigkeitder dynamischen Leistung von Versorgungsspannung
  - Steuerung/Strategien: Softwareunterstützungnotwendig!

Dynamisches Energiemanagement (DPM)- Strategien (Klassen) bestimmt, wann und wie lange eine Hardware-Komponente sich in Energiesparmodusbefinden sollte

- Greedy: Hardware-Komponente sofort nach Erreichen des Leerlaufs in Energiesparmodus, ,,Aufwecken" durch neue Anforderung
- Time-out: Energiesparmodus erst nachdem ein definiertes Intervall im Leerlauf, ,,Aufwecken" wie bei Greedy-Strategien
- Vorhersage: Energiesparmodus sofort nach Erreichen des Leerlaufs, wenn Heuristik vorhersagt,dass Kosten gerechtfertigt
- Stochastisch: Energiesparmodus auf Grundlage eines stochastischen Modells

Spannungsskalierung (DVS)

- Ziel: Unterstützung von DPM-Strategien durch Maßnahmen auf Ebene von Compiler, Betriebssystem und Applikationen:
  - **Compiler**
    - kann Informationen zur Betriebssystem-Unterstützung bezüglich Spannungs-Einstellung in Anwendungs-Code einstreuen,
    - damit zur Laufzeit Informationen über jeweilige Arbeitslast verfügbar
- **Betriebssystem (prädiktives Energiemanagement)**
  - kann Benutzung verschiedener Ressourcen (Prozessor usw.) beobachten
  - kann darüber Vorhersagen tätigen
  - kann notwendigen Performanzbereich bestimmen
- **Anwendungen**
  - können Informationen über jeweils für sie notwendige Performanz liefern
- → Kombination mit energieefizientemScheduling!

Speichereffizienz

- ... heißt: Auslastung des verfügbaren Speichers
  - oft implizit: Hauptspeicherauslastung (memoryfootprint)
  - besonders für kleine/mobile Systeme: Hintergrundspeicherauslastung
- Maße zur Konkretisierung:
  - zeitliche Dimension: Maximum vs. Summe genutzten Speichers?
  - physischer Speicherverwaltung? → Belegungsanteil pAR
  - virtuelle Speicherverwaltung? → Belegungsanteil vAR
- Konsequenzen für Ressourcenverwaltung durch BS:
  - Taskverwaltung (Accounting, Multiplexing, Fairness, ...)
  - Programmiermodell, API (besonders: dynamische Speicherreservierung)
  - Sinnfrage und ggf. Strategien virtueller Speicherverwaltung (VMM)
- Konsequenzen für Betriebssystem selbst:
  - minimaler Speicherbedarfdurch Kernel
  - minimale Speicherverwaltungskosten (durch obige Aufgaben)

Hauptspeicherauslastung

-

Problem: externe Fragmentierung

-
- Lösungen:
  - First Fit, Best Fit, WorstFit, Buddy
  - Relokation
- Kompromissloser Weg: kein Multitasking!

Problem: interne Fragmentierung

-
- Lösung:
  - Seitenrahmengröße verringern
  - Tradeoff: dichter belegte vAR → größere Datenstrukturen für Seitentabellen!
- direkter Einfluss des Betriebssystems auf Hauptspeicherbelegung:
  - → Speicherbedarf des Kernels
  - statische(Minimal-) Größe des Kernels (Anweisungen + Daten)
  - dynamischeSpeicherreservierung durch Kernel
  - bei Makrokernel: Speicherbedarf von Gerätecontrollern (Treibern)!

weitere Einflussfaktoren: Speicherverwaltungskosten

- VMM: Seitentabellengröße → Mehrstufigkeit
- Metainformationen über laufende Programme: Größe von Taskkontrollblöcken(Prozess-/Threaddeskriptoren ...)
- dynamische Speicherreservierung durch Tasks

Beispiel 1: sparsam Prozesskontrollblock (PCB, Metadatenstruktur des Prozessdeskriptors) eines kleinen Echtzeit-Kernels (,,DICK"):

Beispiel 2: eher nicht sparsam Linux Prozesskontrollblock (taskstruct):

Hintergrundspeicherauslastung

Einflussfaktoren des Betriebssystems:

- statische Größe des Kernel-Images, welches beim Bootstrapping gelesen wird
- statische Größe von Programm-Images (Standards wie ELF)
- statisches vs. dynamisches Einbinden von Bibliotheken: Größe von Programmdateien
- VMM: Größe des Auslagerungsbereichs (inkl. Teilen der Seitentabelle!) für Anwendungen
- Modularisierung (zur Kompilierzeit) des Kernels: gezielte Anpassung an Einsatzdomäne möglich
- Adaptivität (zur Kompilier-und Laufzeit) des Kernels: gezielte Anpassung an sich ändernde Umgebungsbedingungen möglich (→ Cassini-Huygens-Mission)

Architekturentscheidungen

- bisher betrachtete Mechanismen: allgemein für alle BS gültig
- ... typische Einsatzgebiete sparsamer BS: eingebettete Systeme
- eingebettetes System: (nach [Manl94])
  - Computersystem, das in ein größeres technisches System, welches nicht zur Datenverarbeitung dient,physisch eingebunden ist.
  - Wesentlicher Bestandteil dieses größeren Systems hinsichtlich seiner Entwicklung, technischer Ausstattung sowie seines Betriebs.
  - Liefert Ausgaben in Form von (menschenlesbaren)Informationen, (maschinenlesbaren)Daten zur Weiterverarbeitung und Steuersignalen.
- BS für eingebettete Systeme: spezielle, anwendungsspezifische Ausprägung der Aufgaben eines ,,klassischen" Universal-BS
  - reduzierter Umfang von HW-Abstraktion, generell: hardwarenähere Ablaufumgebung
  - begrenzte (extrem: gar keine) Notwendigkeit von HW-Multiplexing & -Schutz
- daher eng verwandte NFE: Adaptivitätvon sparsamen BS
- sparsame Betriebssysteme:
  - energieeffizient ˜ geringe Architekturanforderungen an energieintensive Hardware (besonders CPU, MMU, Netzwerk)
  - speichereffizient ˜ Auskommen mit kleinen Datenstrukturen (memory footprint)
- Konsequenz: geringe logische Komplexität des Betriebssystemkerns
- sekundär: Adaptivität des Betriebssystemkerns

Makrokernel (monolithischer Kernel)

- User Space:
  - Anwendungstasks
  - CPU im unprivilegiertenModus (Unix ,,Ringe" 1...3)
  - Isolation von Tasks durch Programmiermodell(z.B. Namespaces) oder VMM(private vAR)
- Kernel Space:
  - Kernelund Gerätecontroller (Treiber)
  - CPU im privilegierten Modus (Unix ,,Ring" 0)
  - keine Isolation (VMM: Kernel wird in alle vAR eingeblendet)

Mikrokernel

- User Space:
  - Anwendungstasks, Kernel-und Treiber tasks (Serverprozesse, grau)
  - CPU im unprivilegiertenModus
  - Isolation von Tasks durch VMM
- Kernel Space:
  - funktional minimaler Kernel(μKernel)
  - CPU im privilegierten Modus
  - keine Isolation (Kernel wird in alle vAR eingeblendet)

Architekturkonzepte im Vergleich

- Makrokernel:
  - ✓vglw. geringe Kosten von Kernelcode (Energie, Speicher)
  - ✓VMM nicht zwingend erforderlich
  - ✓Multitasking (→ Prozessmanagement!)nicht zwingend erforderlich
  - ✗Kernel (inkl. Treibern) jederzeit im Speicher
  - ✗Robustheit, Sicherheit, Adaptivität
- Mikrokernel:
  - ✓Robustheit, Sicherheit, Adaptivität
  - ✓Kernelspeicherbedarf gering, Serverprozesse nur wenn benötigt (→ Adaptivität)
  - ✗hohe IPC-Kosten von Serverprozessen
  - ✗Kontextwechselkosten von Serverprozessen
  - ✗VMM, Multitasking i.d.R. erforderlich

Beispiel-Betriebssysteme

TinyOS

- Beispiel für sparsame BS im Bereich eingebetteter Systeme
- verbreitete Anwendung: verteilte Sensornetze (WSN)
- ,,TinyOS" ist ein quelloffenes, BSD-lizenziertes Betriebssystem
- das für drahtlose Geräte mit geringem Stromverbrauch, wie sie in
  - Sensornetzwerke, (→ Smart Dust)
  - Allgegenwärtiges Computing,
  - Personal Area Networks,
  - intelligente Gebäude,
  - und intelligente Zähler.
- Architektur:
  - grundsätzlich: monolithisch (Makrokernel) mit Besonderheiten:
  - keine klare Trennung zwischen der Implementierung von Anwendungen und BS (wohl aber von deren funktionalen Aufgaben!)
  - → zur Laufzeit: 1 Anwendung + Kernel
- Mechanismen:
  - kein Multithreading, keine echte Parallelität
  - → keine Synchronisation zwischen Tasks
  - → keine Kontextwechsel bei Taskwechsel
  - Multitasking realisiert durch Programmiermodell
  - nicht-präemptives FIFO-Scheduling
  - kein Paging → keine Seitentabellen, keine MMU
- in Zahlen:
  - Kernelgröße: 400 Byte
  - Kernelimagegröße: 1 - 4 kByte
  - Anwendungsgröße: typisch ca. 15 kB, Datenbankanwendung: 64 kB
- Programmiermodell:
  - BS und Anwendung werden als Ganzes übersetzt: statische Optimierungen durch Compilermöglich (Laufzeit, Speicherbedarf)
  - Nebenläufigkeit durch ereignisbasierte Kommunikation zw. Anwendung und Kernel
    - → command: API-Aufruf, z.B. EA-Operation (vglb. Systemaufruf)
    - → event: Reaktion auf diesen durch Anwendung
  - sowohl commands als auch events : asynchron

Beispieldeklaration:

RIOT

[RIOT-Homepage: http://www.riot-os.org]

- ebenfalls sparsames BS,optimiert für anspruchsvollere Anwendungen (breiteres Spektrum)
- ,,RIOT ist ein Open-Source-Mikrokernel-basiertes Betriebssystem, das speziell für die Anforderungen von Internet-of-Things-Geräten (IoT) und anderen eingebetteten Geräten entwickelt wurde."
  - Smartdevices,
  - intelligentes Zuhause, intelligente Zähler,
  - eingebettete Unterhaltungssysteme
  - persönliche Gesundheitsgeräte,
  - intelligentes Fahren,
  - Geräte zur Verfolgung und Überwachung der Logistik.
- Architektur:
  - halbwegs: Mikrokernel
  - energiesparendeKernelfunktionalität:
    - minimale Algorithmenkomplexität
    - vereinfachtes Threadkonzept → keine Kontextsicherung erforderlich
    - keine dynamische Speicherallokation
    - energiesparende Hardwarezustände vom Scheduler ausgelöst (inaktive CPU)
  - Mikrokerneldesign unterstützt komplementäre NFE: Adaptivität, Erweiterbarkeit
  - Kosten: IPC (hier gering!)
- Mechanismen:
  - Multithreading-Programmiermodell
  - modulare Implementierung von Dateisystemen, Scheduler, Netzwerkstack
- in Zahlen:
  - Kernelgröße: 1,5 kByte
  - Kernelimagegröße: 5 kByte

Implementierung

- ... kann sich jeder mal ansehen (keine spezielle Hardware, beliebige Linux-Distribution, FreeBSD, macOSX mit git):
- startet interaktive Instanz von RIOT als ein Prozess des Host-BS
- Verzeichnis RIOT: Quellenzur Kompilierung des Kernels, mitgelieferte Bibliotheken, Gerätetreiber, Beispielanwendungen; z.B.:
  - RIOT/core/include/thread.h: Threadmodell, Threaddeskriptor
  - RIOT/core/include/sched.h,
  - RIOT/core/sched.c: Implementierung des (einfachen) Schedulers
- weitere Infos: riot-os.org/api

Robustheit und Verfügbarkeit

Motivation

- allgemein: verlässlichkeitskritischeAnwendungsszenarien
- Forschung in garstiger Umwelt
- Weltraumerkundung
- hochsicherheitskritische Systeme:
  - Rechenzentren von Finanzdienstleistern
  - Rechenzentren von Cloud-Dienstleistern
- hochverfügbare System:
  - all das bereits genannte
  - öffentliche Infrastruktur(Strom, Fernwärme, ...)
- HPC (high performancecomputing)

Allgemeine Begriffe

- Verlässlichkeit, Zuverlässigkeit (dependability)
- übergeordnete Eigenschaft eines Systems [ALRL04]
- Fähigkeit, eine Leistungzu erbringen, der man berechtigterweise vertrauen kann
- Taxonomie: umfasst entsprechend Definition die Untereigenschaften
  1. Verfügbarkeit (availability)
  2. Robustheit (robustness, reliability
  3. (Funktions-) Sicherheit (safety)
  4. Vertraulichkeit (confidentiality)
  5. Integrität (integrity)
  6. Wartbarkeit (maintainability) (vgl.: evolutionäre Eigenschaften)
- 1., 4. & 5. auch Untereigenschaften von IT-Sicherheit (security)
- → nicht für alle Anwendungen sind alle Untereigenschaften erforderlich

Robustheitsbegriff

- Teil der primären Untereigenschaften von Verlässlichkeit: Robustheit (robustness, reliability)
- Ausfall: beobachtbare Verminderung der Leistung, die ein System tatsächlich erbringt, gegenüber seiner als korrekt spezifizierten Leistung
- Robustheit: Verlässlichkeit unter Anwesenheit externer Ausfälle (= Ausfälle, deren Ursache außerhalb des betrachteten Systems liegt)
- im Folgenden: kurze Systematik der Ausfälle ...

Fehler und Ausfälle ...

- Fehler → fehlerhafter Zustand → Ausfall
- grundlegende Definitionen dieser Begriffe (ausführlich: [ALRL04, AvLR04]):
  - Ausfall (failure): liegt vor, wenn tatsächliche Leistung(en), die ein System erbringt, von als korrekt spezifizierter Leistung abweichen
  - fehlerhafter Zustand (error): notwendige Ursacheeines Ausfalls (nicht jeder error muss zu failure führen)
  - Fehler (fault): Ursache für fehlerhaften Systemzustand (error), z.B. Programmierfehler
-

... und ihre Vermeidung

- Umgang mit ...
  - faults:
    - Korrektheit testen
    - Korrektheit beweisen(→ formale Verifikation)
  - errors:
    - Maskierung, Redundanz
    - Isolationvon Subsystemen
    - → Isolationsmechanismen
  - failures:
    - Ausfallverhalten (neben korrektem Verhalten) spezifizieren
    - Ausfälle zur Laufzeit erkennen und Folgen beheben, abschwächen...
    - → Micro-Reboots

Fehlerhafter Zustand

- interner und externer Zustand (internal & external state)
  - externer Zustand (einer Systems oder Subsystems): der Teil des Gesamtzustands, der an externer Schnittstelle (also für das umgebende (Sub-) System) sichtbar wird
  - interner Zustand: restlicher Teilzustand
  - (tatsächlich) erbrachte Leistung: zeitliche Folge externer Zustände
- Beispiele für das System (Betriebssystem-) Kernel :
  - Subsysteme: Dateisystem, Scheduler, E/A, IPC, ..., Gerätetreiber
  - fault : Programmierfehler im Gerätetreiber
  - externer Zustand des Treibers (oder des Dateisystems, Schedulers, E/A, IPC, ...) ⊆ interner Zustand des Kernels

Fehlerausbreitung und (externer) Ausfall

- Wirkungskette: -[X] Treiber-Programmierfehler (fault) -[X] fehlerhafter interner Zustand des Treibers (error)
  - Ausbreitung dieses Fehlers (failure des Treibers)
  - = fehlerhafter externer Zustand des Treibers
  - = fehlerhafter interner Zustand des Kernels(error)
  - = Kernelausfall!(failure)
- ☒ Auswirkung: fehlerhafter interner Zustand eines weiteren Kernel-Subsystems (z.B. error des Dateisystems)
- → Robustheit: Isolationsmechanismen

Isolationsmechanismen

- im Folgenden: Isolationsmechanismen für robuste Betriebssysteme
  - durch strukturierte Programmierung
  - durch Adressraumisolation
- es gibt noch mehr: Isolationsmechanismen für sichere Betriebssysteme
  - all die obigen...
  - durch kryptografische Hardwareunterstützung: Enclaves
  - durch streng typisierte Sprachen und managed code
  - durch isolierte Laufzeitumgebungen: Virtualisierung

Strukturierte Programmierung

Monolithisches BS... in historischer Reinform:

- Anwendungen
- Kernel
- gesamte BS-Funktionalität
- programmiert als Sammlung von Prozeduren
- jede darf jede davon aufrufen
- keine Modularisierung
- keine definierten internen Schnittstellen

Monolithisches Prinzip

- Ziel: Isolation zwischen Anwendungen und Betriebssystem
- Mechanismus: Prozessor-Privilegierungsebenen (user space und kernel space)
- Konsequenz für Strukturierung des Kernels: Es gibt keine Strukturierung des Kernels ...
- ... jedenfalls fast: Ablauf eines Systemaufrufs (Erinnerung)

Strukturierte Makrokernarchitektur

- Resultat: schwach strukturierter (monolithischer) Makrokernel
- – nach [TaWo05], S. 45
- Weiterentwicklung:
- Schichtendifferenzierung (layered operating system)
- Modularisierung (Bsp.: Linux-Kernel) | Kernelcode | | ---------- | | VFS | | IPC, Dateisystem | | Scheduler, VMM | | Dispatcher, Gerätetreiber |
- Modularer Makrokernel:
- alle Kernelfunktionen in Moduleunterteilt (z.B. verschiedene Dateisystemtypen) → Erweiterbarkeit, Wartbarkeit, Portierbarkeit
- klar definierte Modulschnittstellen(z.B. virtualfilesystem, VFS)
- Module zur Kernellaufzeit dynamisch einbindbar (→ Adaptivität)

Fehlerausbreitung beim Makrokernel

- strukturierte Programmierung:
- ✓Wartbarkeit
- ✓Portierbarkeit
- ✓Erweiterbarkeit
- O (begrenzt) Adaptivität
- O (begrenzt) Schutz gegen statische Programmierfehler: nur durch Compiler (z.B. C private, public)
- ✗kein Schutz gegen dynamische Fehler
- → Robustheit...?
- nächstes Ziel: Schutz gegen Laufzeitfehler... → Laufzeitmechanismen

Adressraumisolation

- zur Erinnerung: private virtuelle Adressräume zweier Tasks ($i\not= j$)
- private virtuelle vs. physischer Adresse

Private virtuelle Adressräume und Fehlerausbreitung

- korrekte private vAR ˜ kollisionsfreie Seitenabbildung!
- Magie in Hardware: MMU (BS steuert und verwaltet...)
- Robustheit: Was haben wir von privaten vAR?
  - ✓nichtvertrauenswürdiger (i.S.v. potenziell nicht korrekter) Code kann keine beliebigen physischen Adressen schreiben (er erreicht sie ja nicht mal...)
  - ✓Kommunikation zwischen nvw. Code (z.B. Anwendungstasks) muss durch IPC-Mechanismen explizit hergestellt werden (u.U. auch shared memory)
    - → Überwachung und Validierung zur Laufzeit möglich
  - ✓Kontrollfluss begrenzen: Funktionsaufrufe können i.A. (Ausnahme: RPC) keine AR-Grenzen überschreiten
    - → BS-Zugriffssteuerungkann nicht durch Taskfehler ausgehebelt werden
    - → unabsichtliche Terminierungsfehler(unendliche Rekursion) erschwert ...

Was das für den Kernel bedeutet

- private virtuelle Adressräume
  - gibt es schon so lange wie VMM
  - gab es lange nur auf Anwendungsebene
  - → keine Isolation zwischen Fehlern innerhalb des Kernels!
- nächstes Ziel: Schutz gegen Kernelfehler (Gerätetreiber)... → BS-Architektur

Mikrokernelarchitektur

- Fortschritt ggü. Makrokernel:
  - Strukturierungskonzept:
    - strenger durchgesetzt durch konsequente Isolation voneinander unabhängiger Kernel-Subsysteme
    - zur Laufzeit durchgesetzt → Reaktion auf fehlerhafte Zustände möglich!
  - zusätzlich zu vertikaler Strukturierung des Kernels: horizontale Strukturierungeingeführt
    - → funktionale Einheiten: vertikal (Schichten)
    - → isolierteEinheiten: horizontal (private vAR)
- Idee:
  - Kernel (alle BS-Funktionalität) → μKernel (minimale BS-Funktionalität)
  - Rest (insbes. Treiber): ,,gewöhnliche" Anwendungsprozesse mit Adressraumisolation
  - Kommunikation: botschaftenbasierteIPC (auch client-server operating system)
  - Nomenklatur: Mikrokernelund Serverprozesse

Modularer Makrokernel vs. Mikrokernel

-
- minimale Kernelfunktionalität:
- keine Dienste, nur allgemeine Schnittstellenfür diese
- keine Strategien, nur grundlegende Mechanismenzur Ressourcenverwaltung
- neues Problem: minimales Mikrokerneldesign
- ,,Wir haben 100 Leute gefragt...": Wie entscheide ich das?
- – Ablauf eines Systemaufrufs
  – schwarz: unprivilegierteInstruktionen
  – blau:privilegierte Instruktionen
  – rot:Übergang zwischen beidem (μKern → Kontextwechsel!)

Robustheit von Mikrokernen

- = Gewinn durch Adressraumisolation innerhalb des Kernels
  - ✓kein nichtvertrauenswürdiger Code im kernelspace , der dort beliebige physische Adressen manipulieren kann
  - ✓Kommunikation zwischen nvw. Code (nicht zur zwischen Anwendungstasks)muss durch IPC explizit hergestellt werden → Überwachung und Validierung zur Laufzeit
  - ✓Kontrollfluss begrenzen: Zugriffssteuerung auch zwischen Serverprozessen, zur Laufzeit unabhängiges Teilmanagement von Code (Kernelcode) möglich (z.B.: Nichtterminierung erkennen)
- Neu:
  - ✓nvw. BS-Code muss nicht mehr im kernelspace (höchste Prozessorprivilegierung) laufen
  - ✓verbleibender Kernel (dessen Korrektheit wir annehmen): klein, funktional weniger komplex, leichter zu entwickeln, zu testen, evtl. formal zu verifizieren
  - ✓daneben: Adaptivitätdurch konsequentere Modularisierung des Kernels gesteigert

Mach

- Mikrokernel-Design: Erster Versuch
  - Carnegie Mellon University (CMU), School of Computer Science 1985 - 1994
- ein wenig Historie
  - UNIX (Bell Labs) - K. Thompson, D. Ritchie
  - BSD (U Berkeley) - W. Joy
  - System V - W. Joy
  - Mach (CMU) - R. Rashid
  - MINIX - A. Tanenbaum
  - NeXTSTEP (NeXT) - S. Jobs
  - Linux - L. Torvalds
  - GNU Hurd (FSF) - R. Stallman
  - Mac OS X (Apple) - S. Jobs

Mach: Ziele Entstehung

- Grundlage:
  - 1975: Aleph(BS des ,,Rochester Intelligent Gateway"), U Rochester
  - 1979/81: Accent (verteiltes BS), CMU
- gefördert durch militärische Geldgeber:
  - DARPA: Defense AdvancedResearch Projects Agency
  - SCI: Strategic Computing Initiative

Ziele

- Mach 3.0 (Richard Rashid, 1989): einer der ersten praktisch nutzbaren μKerne
- Ziel: API-Emulation($\not=$ Virtualisierung!)von UNIX und -Derivaten auf unterschiedlichen Prozessorarchitekturen
- mehrere unterschiedliche Emulatoren gleichzeitig lauffähig
  - Emulation außerhalb des Kernels
  - jeder Emulator:
    - Komponente im Adressraum des Applikationsprogramms
    - 1...n Server, die unabhängig von Applikationsprogramm laufen

Mach-Server zur Emulation

-
- Emulation von UNIX-Systemen mittels Mach-Serverprozessen

μKernel-Funktionen

1. Prozessverwaltung
2. Speicherverwaltung
3. IPC-und E/A-Dienste, einschließlich Gerätetreiber

unterstützte Abstraktionen (→ API, Systemaufrufe):

1. Prozesse
2. Threads
3. Speicherobjekte
4. Ports (generisches, ortstransparentes Adressierungskonzept; vgl. UNIX ,,everything is a file")
5. Botschaften
6. ... (sekundäre, von den obigen genutzte Abstraktionen)

Architektur

-
- Systemaufrufkosten:
  - IPC-Benchmark (1995): i486 Prozessor, 50 MHz
  - Messung mit verschiedenen Botschaftenlängen(x - Werte)
  - ohne Nutzdaten (0 Byte Botschaftenlänge): 115 μs (Tendenz unfreundlich ...)
- Bewertung aus heutiger Sicht:
  - funktional komplex
  - 153 Systemaufrufe
  - mehrere Schnittstellen, parallele Implementierungen für eine Funktion
  - → Adaptivität (Auswahl durch Programmierer)
- Fazit:
  - zukunftsweisender Ansatz
  - langsame und ineffiziente Implementierung

Lessons Learned

- erster Versuch:
- Idee des Mikrokernelsbekannt
- Umsetzung: Designkriterienweitgehend unbekannt
- Folgen für Performanz und Programmierkomfort: [Heis19]
- ✗,,complex"
- ✗,,inflexible"
- ✗,,slow"
- wir wissen etwas über Kosten: IPC-Performanz, Kernelabstraktionen
- wir wissen noch nichts über guten μKern-Funktionsumfangund gute Schnittstellen...
- → nächstes Ziel!

L4

- Made in Germany:
  - Jochen Liedtke (GMD, ,,Gesellschaft für Mathematik und Datenverarbeitung"), Betriebssystemgruppe (u.a.): J. Liedtke, H. Härtig, W. E. Kühnhauser
  - Symposium on Operating Systems Principles 1995 (SOSP '95): ,,On μ-Kernel Construction" [Lied95]
- Analyse des Mach-Kernels:
  1. falsche Abstraktionen
  2. unperformanteKernelimplementierung
  3. prozessorunabhängige Implementierung
     - Letzteres: effizienzschädliche Eigenschaft eines Mikrokernels
     - Neuimplementierung eines (konzeptionell sauberen!) μ-Kerns kaum teurer als Portierung auf andere Prozessorarchitektur

L3 und L4

- Mikrokerne der 2. Generation
- zunächst L3, insbesondere Nachfolger L4: erste Mikrokerne der 2. Generation
- vollständige Überarbeitung des Mikrokernkonzepts: wesentliche Probleme der 1. Generation (z.B. Mach) vermieden
- Bsp.: durchschnittliche Performanz von User-Mode IPC in L3 ggü. Mach: Faktor 22 zugunsten L3
  - heute: verschiedene Weiterentwicklungen von L4 (bezeichnet heute Familie ähnlicher Mikrokerne)

Mikrokernel-Designprinzipien

- Was gehört in einen Mikrokern?
  - Liedtke: Unterscheidung zwischen Konzepten und deren Implementierung
  - bestimmende Anforderungen an beide:
    - Konzeptsicht → Funktionalität,
    - Implementierungssicht → Performanz
  - → 1. μKernel-Generation: Konzept durch Performanzentscheidungen aufgeweicht
  - → Effekt in der Praxis genau gegenteilig: schlechte (IPC-) Performanz!

,,The determining criterion used is functionality, not performance. More precisely, a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the systems's required functionality ." [Jochen Liedtke]

Designprinzipien für Mikrokernel-Konzept:

- → Annahmen hinsichtlich der funktionalen Anforderungen:

1. System interaktive und nicht vollständig vertrauenswürdige Applikationen unterstützen (→ HW-Schutz, -Multiplexing),
2. Hardware mit virtueller Speicherverwaltung und Paging

Designprinzipien:

1. Autonomie: ,,Ein Subsystem (Server)muss so implementiert werden können, dass es von keinem anderen Subsystem gestört oder korrumpiert werden kann."
2. Integrität: ,,Subsystem (Server) S_1 muss sich auf Garantien von S_2 verlassen können. D.h. beide Subsysteme müssen miteinander kommunizieren können, ohne dass ein drittes Subsystem diese Kommunikation stören, fälschen oder abhören kann."

L4: Speicherabstraktion

- Adressraum: Abbildung, die jede virtuelle Seite auf einen physischen Seitenrahmen abbildet oder als ,,nicht zugreifbar" markiert
- Implementierung über Seitentabellen, unterstützt durch MMU-Hardware
- Aufgabe des Mikrokernels (als gemeinsame obligatorische Schicht aller Subsysteme): muss Hardware-Konzept des Adressraums verbergen und durch eigenes Adressraum-Konzept überlagern (sonst Implementierung von VMM-Mechanismen durch Server unmöglich)
- Mikrokernel-Konzept des Adressraums:
  - muss Implementierung von beliebigen virtuellen Speicherverwaltungs-und -schutzkonzepten oberhalb des Mikrokernels (d.h. in den Subsystemen) erlauben
  - sollte einfach und dem Hardware-Konzept ähnlich sein
- Idee: abstrakte Speicherverwaltung
  - rekursive Konstruktion und Verwaltung der Adressräume auf Benutzer-(Server-)Ebene
  - Mikrokernel stellt dafür genau drei Operationen bereit:
    1. grant(x) - Server S überträgt Seite x seines AR in AR von Empfänger S'
    2. map(x) - Server S bildet Seite x seines AR in AR von Empfänger S' ab
    3. flush(x) - Server S entfernt (flusht) Seite x seines AR aus allen fremden AR

Hierarchische Adressräume

- Rekursive Konstruktion der Adressraumhierarchie
  - Server und Anwendungenkönnen damit ihren Klienten Seiten des eigenen Adressraumes zur Verfügung stellen
  - Realspeicher: Ur-Adressraum, vom μKernel verwaltet
  - Speicherverwaltung(en), Paging usw.: vollständig außerhalb des μ-Kernels realisiert

L4: Threadabstraktion

- Thread
  - innerhalb eines Adressraumesablaufende Aktivität
  - → Adressraumzuordnung ist essenziell für Threadkonzept (Code + Daten)
    - Bindung an Adressraum: dynamisch oder fest
    - Änderung einer dynamischen Zuordnung: darf nur unter vertrauenswürdiger Kontrolle erfolgen (sonst: fremde Adressräume les- und korrumpierbar)
- Designentscheidung
  - → Autonomieprinzip
  - → Konsequenz: Adressraumisolation
  - → entscheidender Grund zur Realisierung des Thread-Konzepts innerhalb des Mikrokernels

IPC

- Interprozess-Kommunikation
  - Kommunikation über Adressraumgrenzen: vertrauenswürdig kontrollierte Aufhebung der Isolation
  - → essenziell für (sinnvolles) Multitasking und -threading
- Designentscheidung
  - → Integritätsprinzip
  - → wir haben schon: vertrauenswürdige Adressraumisolation im μKernel
  - → grundlegendes IPC-Konzepts innerhalb des Mikrokernels (flexibel und dynamisch durch Server erweiterbar, analog Adressraumhierarchie)

Identifikatoren

- Thread-und Ressourcenbezeichner
  - müssen vertrauenswürdig vergeben (authentisch und i.A. persistent) und verwaltet(eindeutig und korrekt referenzierbar)werden
  - → essenziell für (sinnvolles) Multitasking und -threading
  - → essenziell für vertrauenswürdige Kernel-und Server-Schnittstellen
- Designentscheidung
  - → Integritätsprinzip
  - → ID-Konzept innerhalb des Mikrokernels (wiederum: durch Server erweiterbar)

Lessons Learned

1. Ein minimaler Mikrokernel
   - soll Minimalmenge an geeigneten Abstraktionenzur Verfügung stellen:
   - flexibel genug, um Implementierung beliebiger Betriebssysteme zu ermöglichen
   - Nutzung umfangreicher Mengeverschiedener Hardware-Plattformen
2. Geeignete, funktional minimale Mechanismen im μKern:
   - Adressraum mit map-, flush-, grant-Operation

- Threadsinklusive IPC
- eindeutige Identifikatoren

3. Wahl der geeigneten Abstraktionen:
   - kritischfür Verifizierbarkeit (→ Robustheit), Adaptivität und optimierte Performanz des Mikrokerns
4. Bisherigen μ-Kernel-Abstraktionskonzepte:
   1. ungeeignete
   2. zu viele
   3. zu spezialisierte u. inflexible Abstraktionen
5. Konsequenzen für Mikrokernel-Implementierung
   - müssen für jeden Prozessortyp neu implementiert werden
   - sind deshalb prinzipiell nicht portierbar → L3-und L4-Prototypen by J. Liedtke: 99% Assemblercode
6. innerhalb eines Mikrokernels sind
   1. grundlegende Implementierungsentscheidungen
   2. meiste Algorithmen u. Datenstrukturen
   - von Prozessorhardware abhängig

- Fazit:
  - Mikrokernelmit akzeptabler Performanz: hardwarespezifische Implementierung minimalerforderlicher, vom Prozessortyp unabhängiger Abstraktionen

Heutige Bedeutung

- nach Tod von J. Liedtke (2001) auf Basis von L4 zahlreiche moderne BS
- L4 heute: Spezifikation eines Mikrokernels (nicht Implementierung)
-
- Einige Weiterentwicklungen:
- TU Dresden (Hermann Härtig): Neuimplementierung in C++ (L4/Fiasco), Basis des Echtzeit-Betriebssystems DROPS, der VirtualisierungsplattformNOVA (genauer: Hypervisor) und des adaptiven BS-Kernels Fiasco.OC
- University ofNew South Wales (UNSW), Australien (Gernot Heiser):
  - Implementierung von L4 auf verschiedenen 64 - Bit-Plattformen, bekannt als L4/MIPS, L4/Alpha
  - Implementierung in C (Wartbarkeit, Performanz)
  - Mit L4Ka::Pistachio bisher schnellste Implementierung von botschaftenbasierterIPC (2005: 36 Zyklen auf Itanium-Architektur)
  - seit 2009: seL4, erster formal verifizierter BS-Kernel (d.h. mathematisch bewiesen, dass Implementierung funktional korrekt ist und nachweislich keinen Entwurfsfehler enthält)

Zwischenfazit

- Begrenzung von Fehlerausbreitung (→ Folgen von errors):
- konsequent modularisierte Architektur aus Subsystemen
- Isolationsmechanismen zwischen Subsystemen
- Konsequenzen für BS-Kernel:
- statische Isolation auf Quellcodeebene → strukturierte Programmierung
- dynamische Isolation zur Laufzeit → private virtuelle Adressräume
- Architektur, welche diese Mechanismen komponiert: Mikrokernel
- Was haben wir gewonnen?
- ✓Adressraumisolation für sämtlichen nichtvertrauenswürdigen Code
- ✓keine privilegierten Instruktionen in nvw. Code (Serverprozesse)
- ✓geringe Größe (potenziell: Verifizierbarkeit) des Kernels
- ✓neben Robustheit: Modularitätund Adaptivitätdes Kernels
- Und was noch nicht?
  - ✗Behandlung von Ausfällen (→ abstürzende Gerätetreiber ...)

3.5 Micro-Reboots

- Beobachtungen am Ausfallverhalten von BS:
- Kernelfehler sind (potenziell) fatal für gesamtes System
- Anwendungsfehler sind es nicht
- → kleiner Kernel = geringeres Risiko von Systemausfällen
- → durch BS-Code in Serverprozessen: verbleibendes Risiko unabhängiger Teilausfälle von BS-Funktionalität (z.B. FS, Treiberprozesse, GUI, ...)
- Ergänzung zu Isolationsmechanismen:
- Mechanismen zur Behandlung von Subsystem-Ausfällen
- = Mechanismen zur Behandlung Anwendungs-, Server- und Gerätetreiberfehlen
- → Micro-Reboots

Ansatz

- wir haben:
- kleinen, ergo vertrauenswürdigen (als fehlerfrei angenommenen)μKernel
- BS-Funktionalität in bedingt vertrauenswürdigen Serverprozessen (kontrollierbare, aber wesentlich größere Codebasis)
- Gerätetreiber und Anwendungen in nicht vertrauenswürdigen Prozessen (nicht kontrollierbare Codebasis)
- wir wollen:
- Systemausfälle verhindern durch Vermeidung von errors im Kernel → höchste Priorität
- Treiber-und Serverausfälle minimieren durch Verbergen ihrer Auswirkungen → nachgeordnete Priorität (Best-Effort-Prinzip)
- Idee:
  - Systemausfälle → μKernel
  - Treiber-und Serverausfälle → Neustart durch spezialisierten Serverprozess

Beispiel: Ethernet-Treiberausfall

-
- schwarz: ausfallfreie Kommunikation
- rot: Ausfall und Behandlung
- blau: Wiederherstellung nach Ausfall

Beispiel: Dateisystem-Serverausfall

-
- schwarz: ausfallfreie Kommunikation
- rot: Ausfall und Behandlung
- blau: Wiederherstellung nach Ausfall

Beispiel-Betriebssystem: MINIX

- Ziele:
- robustes Betriebssystems
- → Schutz gegen Sichtbarwerden von Fehlern(= Ausfälle) für Nutzer
- Fokus auf Anwendungsdomänen: Endanwender-Einzelplatzrechner (Desktop, Laptop, Smart*) und eingebettete Systeme
- Anliegen: Robustheit > Verständlichkeit > geringer HW-Bedarf
- aktuelle Version: MINIX 3.3.0

Architektur

- Kommunikationsschnittstellen ...
  - ... für Anwendungen (weiß): Systemaufrufe im POSIX-Standard
  - ... für Serverprozesse (grau):
    - untereinander: IPC (botschaftenbasiert)
    - mit Kernel: spezielle MINIX-API (kernel calls), für Anwendungsprozesse gesperrt
- Betriebssystem-Serverprozesse:
-
- Dateisystem (FS)
- Prozessmanagement (PM)
- Netzwerkmanagement (Net)
- Reincarnation Server (RS) → Micro-Reboots jeglicher Serverprozesse
- (u. a.) ...
- Kernelprozesse:
- systemtask
- clocktask

Reincarnation Server

- Implementierungstechnik für Micro-Reboots:
- Prozesse zum Systemstart (→ Kernel Image): system, clock, init, rs
  - system, clock: Kernelprogramm
  - init: Bootstrapping (Initialisierung von rs und anderer BS-Serverprozesse), Fork der Login-Shell (und damit sämtlicher Anwendungsprozesse)
  - rs: Fork sämtlicher BS-Serverprozesse, einschließlich Gerätetreiber
-

MINIX: Ausprobieren

- ausführliche Dokumentation
- vorkompiliertes Kernel-Image zum Installieren (VirtualBox, VMWare, ...)

Verfügbarkeit

- komplementäre NFE zu Robustheit: Verfügbarkeit (availability)
  - Zur Erinnerung: Untereigenschaften von Verlässlichkeit
  1. Verfügbarkeit (availability)
  2. Robustheit (robustness, reliability)
- Beziehung:
  - Verbesserung von Robustheit → Verbesserung von Verfügbarkeit
  - Robustheitsmaßnahmen hinreichend , nicht notwendig (hochverfügbare Systeme können sehr wohl von Ausfällen betroffen sein...)
- eine weitere komplementäre NFE:
  - Robustheit → Sicherheit (security)

Allgemeine Definition: Der Grad, zu welchem ein System oder eine Komponente funktionsfähig und zugänglich (erreichbar) ist,wann immer seine Nutzung erforderlichist. (IEEE)
genauer quantifiziert:

- Der Anteil an Laufzeit eines Systems, in dem dieses seine spezifizierte Leistung erbringt.
-
- Availability= Total Uptime/ Total Lifetime= MTTF / (MTTF + MTTR)
  - MTTR: Mean Time to Recovery ... Erwartungswert für TTR
  - MTTF: Mean Time to Failure ... Erwartungswert für TTF
- einige Verfügbarkeitsklassen: | Verfügbarkeit | Ausfallzeit pro Jahr | Ausfallzeit pro Woche | | ------------ | ------------------- | ------------------- | | 90% | > 1 Monat | ca. 17 Stunden | | 99% | ca. 4 Tage | ca. 2 Stunden | | 99,9% | ca. 9 Stunden | ca. 10 Minuten | | 99,99% | ca. 1 Stunde | ca. 1 Minute | | 99,999% | ca. 5 Minuten | ca. 6 Sekunden | | 99,9999% | ca. 2 Sekunden | << 1 Sekunde |
- Hochverfügbarkeitsbereich (gefeierte ,,five nines" availability)
- Maßnahmen:
- Robustheitsmaßnahmen
- Redundanz
- Ausfallmanagement

QNX Neutrino: Hochverfügbares Echtzeit-BS

Überblick QNX:

- Mikrokern-Betriebssystem
- primäres Einsatzfeld: eingebettete Systeme, z.B. Automobilbau
- Mikrokernarchitektur mit Adressraumisolation für Gerätetreiber
- (begrenzt) dynamische Micro-Rebootsmöglich
- → Maximierung der Uptime des Gesamtsystems

Hochverfügbarkeitsmechanismen:

1. ,,High-Avalability-Manager": Laufzeit-Monitor, der Systemdienste oder Anwendungsprozesse überwacht und neustartet → μReboot-Server
2. ,,High-Availability-Client-Libraries": Funktionen zur transparenten automatischen Reboot für ausgefallene Server-Verbindungen

Sicherheit

Motivation

Medienberichte zu IT-Sicherheitsvorfällen:

- 27.-28.11.2016: Ausfälle von über 900.000 Kundenanschlüssen der Deutschen Telekom
  - Bundesamt für Sicherheit in der Informationstechnik (BSI): weltweiter Angriff auf ausgewählte Fernverwaltungsports von DSL-Routern, um angegriffene Geräte mit Schadsoftware zu infizieren
  - Angreiferziel: Missbrauch der Hardware für eigentliche Angriffe (Botnet)
- 15.05.-06.06.2019: Ransomware-Angriff zur Erpressung der Heise Verlagsgruppe
  - Infektion eines Rechners im lokalen Netz durch Malware in eMail-Anhang (Trojaner)
  - Täuschung des Nutzers: Schadcode mit Administratorrechten ausgeführt (Spezialfall von Malware: *Root Kit*)
  - Malwareziel: Verschlüsselungvon Nutzerdaten
  - Angreiferziel: Erspressungvon Lösegeld für Entschlüsselung

Was sichere Betriebssysteme erreichen können ... und was nicht: youtube

Terminologie

Achtung zwei unterschiedliche ,,Sicherheiten"

1. Security (IT-Sicherheit, Informationssicherheit)
   - Ziel: Schutz **des** Rechnersystems
   - hier besprochen
   - Systemsicherheit
2. Safety (Funktionale Sicherheit, Betriebssicherheit)
   - Ziel: Schutz **vor** einem Rechnersystem
   - an dieser Stelle nicht besprochen

Eine (unvollständige) Taxonomie:

-

Sicherheitsziele

Allgemeines Ziel von IT-Sicherheit i.S.v. Security ... ein Rechnersystem sicher zu machen gegen Schäden durch zielgerichtete Angriffe, insbesondere in Bezug auf die Informationen, die in solchen Systemen gespeichert, verarbeitet und übertragen werden. (Programme sind somit ebenfalls als Informationen zu verstehen.)
Cave! Insbesondere für Sicherheitsziele gilt: Daten $\not=$ Informationen
Sicherheitsziele: sukzessive Konkretisierungen dieser Allgemeinformel hinsichtlich anwendungsspezifischer Anforderungen

Abstrakte Ziele:

1. Vertraulichkeit (Confidentiality)
2. Integrität (Integrity)
3. Verfügbarkeit (Availability)
4. Authentizität (Authenticity)
5. Verbindlichkeit = Nichtabstreitbarkeit (Non-repudiability)

Abstrakte Ziele dienen zur Ableitung konkreter Sicherheitsziele. Wir definieren sie als Eigenschaften von gespeicherten oder übertragenen Informationen ...

- Vertraulichkeit: ... nur für einen autorisierten Nutzerkreis zugänglich (i.S.v. interpretierbar) zu sein.
- Integrität: ... vor nicht autorisierter Veränderung geschützt zu sein.
- Verfügbarkeit: ... autorisierten Nutzern in angemessener Frist zugänglich zu sein.
- Authentizität: ... ihren Urheber eindeutig erkennen zu können.
- Verbindlichkeit: ... sowohl integer als auch authentisch zu sein.

Schadenspotenzial

1. Vandalismus, Terrorismus
   - reine Zerstörungswut
2. Systemmissbrauch
   - illegitime Ressourcennutzung, Ziel i.d.R.: hocheffektive Folgeangriffe
   - Manipulationvon Inhalten (→ Desinformation)
3. (Wirtschafts-) Spionage und Diebstahl
   - Verlust der Kontrolle über kritisches Wissen (→ Risikotechnologien)
   - immense wirtschaftliche Schäden (→ Technologieführer, Patentinhaber)
   - z.B. Diebstahl von industriellem Know-How
4. Betrug, persönliche Bereicherung
   - wirtschaftliche Schäden
5. Sabotage, Erpressung
   - Außerkraftsetzen lebenswichtiger Infrastruktur (z.B. schon Registrierkassen)
   - Erpressung von ausgewählten (oder schlicht großen) Zielgruppen durch vollendete, reversible Sabotage (→ Verschlüsselung von Endanwenderinformationen)

Bedrohungen

1. Eindringlinge (intruders)
   - im engeren Sinne menschliche Angreifer (,,Hacker"), deren Angriff eine technische Schwachstelleausnutzt (exploit)
2. Schadsoftware (malicious software, malware)
   - durch Ausnutzung einer (auch menschlichen) Schwachstelle zur Ausführung gebrachte Programme, die (teil-) automatisierte Angriffe durchführen
   - Trojanische Pferde (trojan horses): scheinbar nützliche Software, die verborgene Angriffsfunktionalität enthält
   - Viren, Würmer (viruses, worms): Schadsoftware, die Funktionalität zur eigenen Vervielfältigung und/oder Modifikation beinhaltet
   - Logische Bomben (logicbombs): Code-Sequenz in trojanischen Pferden, deren Aktivierung an System-oder Datumsereignisse gebunden ist
   - Root Kits
3. Bots und Botnets
   - (weit-) verteilt ausgeführte Schadsoftware
   - eigentliches Ziel i.d.R. nicht das jeweils infizierte System

Professionelle Malware: Root Kit

- Programm-Paket, das unbemerkt Betriebssystem (und ausgewählte Anwendungen) modifiziert, um Administratorrechte zu erlangen
  - Administrator-bzw. Rootrechte: ermöglichen Zugriff auf alle Funktionen und Dienste eines Betriebssystems
  - Angreifer erlangt vollständige Kontrolle des Systems und kann
    - Dateien (Programme) hinzufügen bzw. ändern
    - Prozesse überwachen
    - über die Netzverbindungen senden und empfangen
    - bei all dem Hintertüren für Durchführung und Verschleierung zukünftiger Angriffe platziere
  - Ziele eines Rootkits:
    - seine Existenz verbergen
    - zu verbergen, welche Veränderungen vorgenommen wurden
    - vollständige und irreversible Kontrolle über BS zu erlangen
- Ein erfolgreicher Root-Kit-Angriff ...
  - ... kann jederzeit
  - ... mit hochaktuellem und systemspezifischem Wissen über Schwachstellen
  - ... vollautomatisiert, also reaktiv unverhinderbar
  - ... unentdeckbar
  - ... nicht reversibel
  - ... die uneingeschränkte Kontrolle über das Zielsystem erlangen.
- Voraussetzung: eine einzige Schwachstelle...

Schwachstellen

1. Passwort (begehrt: Administrator-Passwörter...)
   - ,,erraten"
   - zu einfach, zu kurz, usw.
   - Brute-Force-Angriffe mit Rechnerunterstützung
   - Abfangen (eavesdropping)
   - unverschlüsselte Übertragung (verteilte Systeme) oder Speicherung
2. Programmierfehler (Speicherfehler...!)
   - im Anwenderprogrammen
   - in Gerätemanagern
   - im Betriebssystem
3. Mangelhafte Robustheit
   - keine Korrektur fehlerhafter Eingaben
   - buffer overrun/underrun (,, Heartbleed")
4. Nichttechnische Schwachstellen
   - physisch, organisatorisch, infrastrukturell
   - menschlich (→ Erpressung, socialengineering)

Zwischenfazit

- Schwachstellen sind unvermeidbar
- Bedrohungen sind unkontrollierbar
  - ... und nehmen tendeziellzu!

Beides führt zu operationellen Risiken beim Betrieb eines IT-Systems → Aufgabe der Betriebssystemsicherheit: Auswirkungen operationeller Risiken reduzieren (wo diese nicht vermieden werden können...)
Wie dies geht: Security Engineering

Sicherheitspolitiken

- Herausforderung: korrekte Durchsetzung von Sicherheitspolitiken
- Vorgehensweise: Security Engineering

| | | ---------- | ---------- | | Sicherheitsziele | Welche Sicherheitsanforderungen muss das Betriebssystem erfüllen? | | Sicherheitspolitik | Durch welche Strategien soll es diese erfüllen? (→ Regelwerk) | | Sicherheitsmechanismen | Wie implementiert das Betriebssystem seine Sicherheitspolitik? | | Sicherheitsarchitektur | Wo implementiert das Betriebssystem seine Sicherheitsmechanismen (und deren Interaktion)? |

Sicherheitspolitiken und -modelle

Kritischfür korrekten Entwurf, Spezifikation, Implementierung der Betriebssystem-Sicherheitseigenschaften!
Begriffsdefinitionen:

- Sicherheitspolitik (Security Policy): Eine Menge von Regeln, die zum Erreichen eines Sicherheitsziels dienen.
- Sicherheitsmodell (Security Model): Die formale Darstellung einer Sicherheitspolitik zum Zweck
  - der Verifikation ihrer Korrektheit
  - der Spezifikation ihrer Implementierung.

Zugriffssteuerungspolitiken

... geben Regeln vor, welche durch Zugriffssteuerungsmechanismen in BS durchgesetzt werden müssen.
Zugriffssteuerung (access control): Steuerung, welcher Nutzer oder Prozess mittels welcher Operationen auf welche BS-Ressourcen zugreifen darf (z.B.: Anwender darf Textdateien anlegen, Administrator darf Dateisysteme montieren und System-Logdateien löschen, systemd - Prozess darf Prozessdeskriptoren manipulieren, ...)
Zugriffssteuerungspolitik: konkrete Regeln, welche die Zugriffssteuerung in einem BS beschreiben
Zugriffssteuerungsmodell: Sicherheitsmodell einer Zugriffssteuerungspolitik
Zugriffssteuerungsmechanismus: Implementierung einer Zugriffssteuerungspolitik

Beispiele für BS-Zugriffssteuerungspolitiken

klassifiziert nach Semantik der Politikregeln:

- IBAC (Identity-basedAccess Control): Politik spezifiziert, welcher Nutzer an welchen Ressourcen bestimmte Rechte hat.
  - Bsp.: ,,Nutzer Anna darf Brief.docx lesen, aber nicht schreiben."
- TE (Type-Enforcement): Politik spezifiziert Rechte durch zusätzliche Abstraktion (Typen): welcher Nutzertyp an welchem Ressourcentyp bestimmte Rechte hat.
  - Bsp.: ,,Nutzer vom Typ Administrator dürfen Dateien vom Typ Log lesen und schreiben."
- MLS (Multi-Level Security): Politik spezifiziert Rechte, indem aus Nutzern und Ressourcen hierarchische Klassen (Ebenen, ,,Levels") gleicher Kritikalität im Hinblick auf Sicherheitsziele gebildet werden.
  - Bsp.: ,,Nutzer der Klasse nicht vertrauenswürdig dürfen Dateien der Klasse vertraulich nicht lesen."
- DAC (Discretionary Access Control, auch: wahlfreie Zugriffssteuerung): Aktionen der Nutzer setzen die Sicherheitspolitik (oder wesentliche Teile davon) durch. Typisch: Begriff des Eigentümers von BS-Ressourcen.
  - Bsp.: ,,Der Eigentümer einer Datei bestimmt (bzw. ändert), welcher Nutzer welche Rechte daran hat."
- MAC (MandatoryAccess Control, auch: obligatorische Zugriffssteuerung): Keine Beteiligung der Nutzer an der Durchsetzungeiner (zentral administrierten) Sicherheitspolitik.
  - Bsp.: ,,Anhand ihres Dateisystempfads bestimmt das Betriebssystem, welcher Nutzer welche Rechte an einer Datei hat."

Einige Beispiele ...

... und ein Verdacht Eindruck der Effektivität von DAC: ,,[...] so the theory goes. By extension, yes, there may be less malware, but that will depend on whether users keep UAC enabled, which depends on whether developers write software that works with it and that users stop viewing prompts as fast-clicking exercises and actually consider whether an elevation request is legitimate." (Jesper M. Johansson, TechNet Magazine) [https://technet.microsoft.com/en-us/library/2007.09.securitywatch.aspx, Stand: 10.11.2017]

Traditionell: DAC, IBAC

Auszug aus der Unix-Sicherheitspolitik:

- es gibt Subjekte (Nutzer, Prozesse) und Objekte (Dateien, Sockets ...)
- jedes Objekt hat einen Eigentümer
- Eigentümer legen Zugriffsrechte an Objekten fest (→ DAC)
- es gibt drei Zugriffsrechte: read, write, execute
- je Objekt gibt es drei Klassen von Subjekten, für die individuell Zugriffsrechte vergeben werden können: Eigentümer (,,u"), Gruppe (,,g"), Rest der Welt (,,o")

In der Praxis:

- identitätsbasierte (IBAC), wahlfreie Zugriffssteuerung (DAC)
- hohe individuelle Freiheit der Nutzer bei Durchsetzung der Politik
- hohe Verantwortung (,,Welche Nutzer werden jemals in Gruppe vsbs sein...?")

Modellierung: Zugriffsmatrix

- acm (access control matrix): Momentaufnahme der globalen Rechteverteilung zu einem definierten ,,Zeitpunkt t"
- Korrektheitskriterium: Wie kann sich dies nach t möglicherweise ändern...? (HRU-Sicherheitsmodell)[HaRU76]

Modellkorrektheit: Rechteausbreitung

- Änderungsbeispiel: kühnhauser nimmt krause in Gruppe vsbs auf ...
- Rechteausbreitung (privilegeescalation), hier verursacht durch eine legale Nutzeraktion (→ DAC)
  - (Sicherheitseigenschaft: HRU Safety , → ,,Systemsicherheit")

Modern: MAC, MLS

Sicherheitspolitik der Windows UAC (user account control):

- es gibt Subjekte (Prozesse) und Objekte (Dateisystemknoten)
- jedem Subjekt ist eine Integritätsklasse zugewiesen:
  - Low: nicht vertrauenswürdig (z.B. Prozesse aus ausführbaren Downloads)
  - Medium: reguläre Nutzerprozesse, die ausschließlich Nutzerdaten manipulieren
  - High: Administratorprozesse, die Systemdaten manipulieren können
  - System: (Hintergrund-) Prozesse, die ausschließlich Betriebssystemdienste auf Anwenderebene implementieren (etwa der Login-Manager)
- jedem Objekt ist analog eine dieser Integritätsklassen zugewiesen (Kritikalität von z.B. Nutzerdaten vs. Systemdaten)
- sämtliche DAC-Zugriffsrechte (die gibt es auch) müssen mit einer Hierarchie der Integritätsklassen konsistent sein (→ ein bisschen MAC)
- Nutzer können diese Konsistenzanforderung selektiv außer Kraft setzen (→ DAC)

MAC-Modellierung: Klassenhierarchie

Beispiel: Modelliert durch Relation $\leq$: gleich oder kritischer als $\leq={(High , Medium), (High , Low), (Medium , Low), (High , High), (Medium , Medium), (Low , Low)}$

- repräsentiert Kritikalität hinsichtlich des Sicherheitsziels Integrität (Biba-Sicherheitsmodell) [Biba77]
- wird genutzt, um legale Informationsflüsse zwischen Subjekten und Objekten zu modellieren → Schutz vor illegalem Überschreiben
- leitet Zugriffsrechte aus Informationsflüssen ab:
  - Prozess Datei: schreiben
  - Prozess Datei: lesen

DAC-Modellierung: Zugriffsmatrix

Modellkorrektheit: Konsistenz

- Korrektheitskriterium: Garantiert die Politik, dass acm mit $\leq$ jederzeit konsistent ist? (BLP Security) [BeLa76]
- elevation-Mechanismus: verändert nach Nutzeranfrage (→ DAC) sowohl acm als auch $\leq\rightarrow$ konsistenzerhaltend?
- andere BS-Operationen: verändern unmittelbar nur acm (z.B. mittels Dateisystemmanagement) → konsistenzerhaltend?

Autorisierungsmechanismen

Begriffsdefinitionen:

- Sicherheitsmechanismen: Datenstrukturen und Algorithmen, welche die Sicherheitseigenschaften eines Betriebssystems implementieren.
  - → Sicherheitsmechanismen benötigt man zur Herstellung jeglicher Sicherheitseigenschaften (auch jener, die in unseren Modellen implizit angenommen werden!)
  - Nutzerauthentisierung (login - Dientsprogramm, Passwort-Hashing, ...)
  - Autorisierungsinformationen (Metainformationen über Rechte, MLS-Klassen, TE-Typen, ...)
  - Autorisierungsmechanismen (Rechteprüfung, Politikadministration, ...)
  - kryptografische Mechanismen (Verschlüsselungsalgorithmen, Hashfunktionen, ...)
- Auswahl im Folgenden: Autorisierungsmechanismen und -informationen

Traditionell: ACLs, SUID

Autorisierungsinformationen:

- müssen Subjekte (Nutzer) bzw. Objekte (Dateien, Sockets ...) mit Rechten assoziieren → Implementierung der Zugriffsmatrix (acm), diese ist:
  - groß (→ Dateianzahl auf Fileserver)
  - dünn besetzt
  - in Größe und Inhalt dynamisch veränderlich
  - → effiziente Datenstruktur?
- Lösung: verteilte Implementierung der acm als Spaltenvektoren, deren Inhalt in den Objekt-Metadaten repräsentiert wird: Zugriffssteuerungslisten (Access Control Lists , ACLs)

ACLs: Linux-Implementierung

- objektspezifischer Spaltenvektor = Zugriffssteuerungsliste
- Dateisystem-Metainformationen: implementiert in I-Nodes

ACLs: Linux-Implementierung

Modell einer Unix acm ...
	lesen	schreiben	ausführen
Eigentümer (,,u")	ja	ja	ja
Rest der Welt (,,o")	ja	nein	ja
Gruppe (,,g")	ja	nein	ja

- 3 - elementige Liste
- 3 - elementige Rechtemenge
- → 9 Bits
- dessen Implementierung kodiert in 16-Bit-Wort: 1 1 1 1 0 1 1 0 1
- ... und dessen Visualisierung in Linux:

Autorisierungsmechanismen: ACL-Auswertung

Subjekte = Nutzermenge eines Linux-Systems... besteht aus Anzahl registrierter Nutzer

- jeder hat eindeutige UID (userID), z.B. integer- Zahl
- Dateien, Prozesse und andere Ressourcenwerden mit UID des Eigentümersversehen
  - bei Dateien: Teil des I-Nodes
  - bei Prozessen: Teil des PCB (vgl. Grundlagen ,,Betriebssysteme")
  - standardmäßiger Eigentümer: derjenige, eine Ressource erzeugt hat

Nutzergruppen (groups)

- jeder Nutzer wird durch Eintrag in einer Systemdatei (/etc/group) einer oder mehreren Gruppen zugeordnet(→ ACL: ,, g " Rechte)

Superuser oder root... hat grundsätzlich uneingeschränkte Rechte.

- UID = 0
- darf insbesondere alle Dateien im System lesen, schreiben, ausführen; unabhängig von ACL

ACL-Implementierung

- ACLs:
  - in welchen Kerneloperationen?
  - welche Kernelschnittstellen (Rechte prüfen, ändern)?
  - welche Datenstrukturen, wo gespeichert?
- acm und ACLs:
  - Vorteile der Listenimplementierung?
  - Nachteile ggü. zentral implementierter Matrix? (DAC vs. MAC, Administration, Analyse ...)
- → Übung 2

Nutzerrechte → Prozessrechte

bisher:
Linux-Sicherheitspolitik formuliert Nutzerrechte an Dateien (verteilt gespeichert in ACLs)
Durchsetzung: basiert auf Prozessrechten

- Annahme: Prozesse laufen mit UID des Nutzers, welcher sie gestartet hat und repräsentieren Nutzerintention und Nutzerberechtigungen i.S.d. Sicherheitspolitik
- technisch bedeutet dies: ein Nutzer beauftragt einen anderen Prozess, sich zu dublizieren(fork()) und das gewünschte Programm auszuführen (exec*())
- Vererbungsprinzip:

Autorisierungsmechanismen: Set-UID

konsequente Rechtevererbung:

- Nutzer können im Rahmen der DAC-Politik ACLs manipulieren
- Nutzer können (i.A.) jedoch keine Prozess-UIDs manipulieren
- → und genau so sollte es gem. Unix-Sicherheitspolitik auch sein!

Hintergrund:

- Unix-Philosophie ,, everythingisa file ": BS-Ressourcen wie Sockets, IPC-Instanzen, E/A-Gerätehandler als Datei repräsentiert → identische Schutzmechanismen zum regulären Dateisystem
- somit: Autorisierungsmechanismen zur Begrenzung des Zugriffs auf solche Geräte nutzbar (Bsp.: Zugriffe verschiedener Prozesse auf einem Drucker müssen koordiniert, ggf. eingeschränkt werden)
- dazu muss
  - *root* bzw. zweckgebundener Nutzer Eigentümer des Druckers sein
  - ACL als `rw- --- ---` gesetzt sein

Folge:

- Nutzerprozesse könnten z.B. nicht drucken ...

Lösung: Mechanismus zur Rechtedelegation

- implementiert durch ein weiteres ,,Recht" in ACL: SUID-Bit (,, setUID")
- Programmausführung modifiziert Kindprozess, so dass UID des Programmeigentümers (im Bsp.: root) seine Rechte bestimmt
- Technik: eine von UID abweichende Prozess-Metainformation (→ PCB) effektive UID (eUID) wird tatsächlich zur Autorisierung genutzt
- `-rws rws r-x 1 root root 2 2011-10-01 16:00 print`

Strategie für sicherheitskritische Linux-Programme

- Eigentümer: root
- SUID-Bit: gesetzt
- per eUID delegiert root seine Rechte an genau solche Kindprozesse, die SUID-Programme ausführen
- Folge: Nutzerprozesse können Systemprogramme (und nur diese) ohne permanente root - Rechte ausführen

Weiteres Beispiel: passwd

- ermöglicht Nutzern Ändern des (eigenen) Anmeldepassworts
- Schreibzugriff auf /etc/shadow (Password-Hashes) erforderlich ... Schutz der Integrität anderer Nutzerpasswörter?
- Lösung: '-rws rws r-x 1 root root 1 2005-01-20 10:00 passwd$
- passwd - Programm (und nur dieses!) wird mit root-Rechten ausgeführt ()... und passwd schreibt ja nur unseren eigenen Passwort-Hash)

Beispiel passwd

- Problem: privilegierter Zugriff durch unprivilegierte Anwendung
- Standard Linux Lösung:

Modern: SELinux

- Ursprung
  - Anfang 2000er Jahre: sicherheitsfokussiertes Betriebssystemprojekt für US-amerikanische NSA [LoSm01]
  - Implementierung des (eigentlich)μKernel-Architekturkonzepts Flask
  - heute: Open Source, Teil des mainline Linux Kernels
- Klassische UNIXoide: Sicherheitspolitik fest im Kernel implementiert
  - I-Nodes, PCBs, ACLs, UID, GID, SUID, ...
- Idee SELinux: Sicherheitspolitikals eigene BS-Abstraktion
  - zentrale Datenstruktur für Regeln, die erlaubte Zugriffe auf ein SELinux-System definiert
  - erlaubt Modifikation und Anpassung an verschiedene Sicherheitsanforderungen → NFE Adaptivität ...

SELinux-Sicherheitsmechanismen

BS-Komponenten

- Auswertung der Sicherheitspolitik: Security- Server , implementiert als Linux-Kernelmodul(Technik: LSM, Linux Security Module); → entscheidet über alle Zugriffe auf alle Objekte
- Durchsetzung der Sicherheitspolitik : LSM Hooks (generische Anfrage-Schnittstellen in allen BS-Funktionen)
- Administration der Sicherheitspolitik: geschrieben in Textform, muss zur Laufzeit in Security Server installiert werden

SELinux-Sicherheitspolitik

Repräsentation der Sicherheitspolitik:

- physisch: in spezieller Datei, die alle Regeln enthält (in maschinenlesbarer Binärdarstellung), die der Kernel durchsetzen muss
- diese Datei wird aus Menge von Quelldateien in einer Spezifikationssprache für SELinux-Sicherheitspolitiken kompiliert
- diese ermöglicht anforderungsspezifische SELinux-Politiken: können (und müssen) sich von einem SELinux-System zum anderen wesentlich unterscheiden
- Politik wird während des Boot-Vorgangs in Kernel geladen

Politiksemantik

Regeln einer SELinux-Sicherheitspolitiken, Semantische Konzepte(Auswahl):

- Type Enforcement (TE)
- Typisierung von
  - Subjekten: Prozesse
  - Objekten der Klassen: Dateien, Sockets, EA-Geräteschnittstellen, ...
- Rechte delegation durch Retypisierung(vgl. Unix-SUID!)
-

Autorisierungsinformationen

Security Context:
Respräsentiert SELinux-Autorisierungsinformationen für jedes Objekt:

- Semantik:
  - Prozess bash läuft (momentan) mit Typ `shell_t`
  - Datei shadow hat (momentan) den Typen `shadow_t`.

Autorisierungsregeln

... werden systemweit festgelegt in dessen Sicherheitspolitik (→ MAC):
Access Vector Rules

- definieren Autorisierungsregeln basierend auf Subjek-/Objekttypen
- Zugriffe müssen explizit gewährt werden (default-deny)
- Semantik: Erlaube("allow") ...
  - jedem Prozess mit Typ `shell_t`
  - ausführenden Zugriff (benötigt die Berechtigung {`execute`}),
  - auf Dateien (also Objekte der Klassefile)
  - mit Typ `passwd_exec_t`.

Autorisierungsmechanismen: passwd Revisited

Klassischer Anwendungsfall für SELinux-TE: Passwort ändern
Lösung: Retypisierung bei Ausführung

- Prozess wechselt in einen aufgabenspezifischen Typ `passwd_t`
- → massiv verringertes Missbrauchspotenzial!
-

SELinux: weitere Politiksemantiken

- hier nur gezeigt: Überblick über TE
- außerdem relevant für SELinux-Politiken (und deren Administration...):
  - Einschränkung von erlaubten Typtransitionen (Welches Programm darf mit welchem Typ ausgeführt werden?)
  - weitere Abstraktionsschicht: rollenbasierte Regeln (RBAC)
  - → Schutz gegen nicht vertrauenswürdige Nutzer (vs. nvw. Software)
- Ergebnis:
  - ✓extrem feingranulare, anwendungsspezifische Sicherheitspolitik zur Vermeidung von privilege escalation Angriffen
  - ✓obligatorische Durchsetzung (→ MAC, zusätzlich zu Standard-Unix-DAC)
  - O Softwareentwicklung: Legacy-Linux-Anwendungen laufen ohne Einschränkung, jedoch
  - ✗Politikentwicklung und -administrationkomplex!

Weitere Informationen zu SELinux →

MAC-Mechanismen ala SELinux sind heutzutage in vielerlei Software bereits zu finden:

- Datenbanksoftware (SEPostgreSQL)
- Betriebssysteme für mobile Geräte (FlaskDroid)
- sehr wahrscheinlich: zukünftige, sicherheitsorientierte BS...

Isolationsmechanismen

- bekannt: Isolationsmechanismen für robuste Betriebssysteme
  - strukturierte Programmierung
  - Adressraumisolation
- nun: Isolationsmechanismen für sichere Betriebssysteme
  - all die obigen...
  - kryptografische Hardwareunterstützung: Intel SGX Enclaves
  - sprachbasiert:
    - streng typisierte Sprachen und *managed code* : Microsoft Singularity [HLAA05]
    - speichersichere Sprachen (Rust) + Adressraumisolation (μKernel): RedoxOS
  - isolierte Laufzeitumgebungen: Virtualisierung (Kap. 6)

Intel SGX

- SGX: Software Guard Extensions [CoDe16]
- Ziel: Schutz von sicherheitskritischen Anwendungen durch vollständige, hardwarebasierte Isolation
- → strenggenommen kein BS-Mechanismus: Anwendungen müssen dem BS nicht mehr vertrauen! (AR-Schutz, Wechsel von Privilegierungsebenen, ...)
- Annahmen/Voraussetzungen:
  1. sämtliche Software nicht vertrauenswürdig (potenziell durch Angreifer kontrolliert)
  2. Kommunikation mit dem angegriffenen System nicht vertrauenswürdig (weder vertraulich noch verbindlich)
  3. kryptografische Algorithmen (Verschlüsselung und Signierung) sind vertrauenswürdig, also nicht für den Angreifer zu brechen
  4. Ziel der Isolation: Vertraulichkeit, Integrität und Authentizität(nicht Verfügbarkeit) von Anwendungen (Code) und den durch sie verarbeiteten Informationen

Enclaves

- Idee: geschützter Speicherbereich für Teilmenge der Seiten (Code und Daten) einer Task: Enclave Page Cache (EPC)
- Prozessor (und nur dieser) ver-und entschlüsselt EPC-Seiten
-
- Enclaves: Erzeugung
  - Erzeugen: App. → Syscall → BS-Instruktion an CPU (ECREATE)
  - Seiten hinzufügen: App. → Syscall → BS-Instruktion an CPU (EADD)
    - Metainformationen für jede hinzugefügte Seite als Teil der EPC-Datenstruktur (u.a.: Enklave - ID, Zugriffsrechte, vAR-Adresse)
  - Initialisieren: App. → Syscall → BS-Instruktion an CPU (EINIT)
    - finalisiert gesamten Speicherinhalt für diese Enclave
    - CPU erzeugt Hashwert = eindeutige Signatur des Enclave - Speicherinhalts
    - falls BS bis zu diesem Punkt gegen Integrität der Anwendung verstoßen hat: durch Vergleich mit von dritter Seite generiertem Hashwert feststellbar!
- Enclave - Zustandsmodell (vereinfacht) :
- Zugriff: App. → CPU-Instruktionen in User Mode (EENTER, EEXIT)
  - CPU erfordert, dass EPC-Seiten in vARder zugreifenden Task

SGX: Licht und Schatten

- Einführung 2015 in Skylake - Mikroarchitektur
- seither in allen Modellen verbaut, jedoch nicht immer aktiviert
- Nutzer bislang: Demos und Forschungsprojekte, Unterstützung durch einige Cloud-Anbieter, (noch) keine größeren Märkte erschlossen
- Konzept hardwarebasierter Isolation ...
  - ✓liefert erstmals die Möglichkeit zur Durchsetzung von Sicherheitspolitiken auf Anwendungsebene
  - O setzt Vertrauen in korrekte (und nicht böswillige) Hardwarevoraus
  - O Dokumentation und Entwicklerunterstützung (im Ausbau ...)
  - ✗schützt mittels Enclaves einzelne Anwendungen, aber nicht das System
  - ✗steckt hinsichtlich praktischer Eigenschaften noch in den Anfängen (vgl. μKernel...):
    - Performanz [WeAK18]
    - Speicherkapazität(max. Größe EPC: 128 MiB, davon nur 93 MiBnutzbar)
    - → komplementäre NFE: Speichereffizienz!

Sicherheitsarchitekturen

Sicherheitsarchitektur... ist die Softwarearchitektur (Platzierung, Struktur und Interaktion) der Sicherheitsmechanismen eines IT-Systems.

- Voraussetzung zum Verstehen jeder Sicherheitsarchitektur:
  - Verstehen des Referenzmonitorprinzips
  - frühe Forschungen zu Betriebssystemsicherheit in 1970er-1980er Jahren durch US-Verteidigungsministerium
  - Schlüsselveröffentlichung: Anderson-Report(1972)[Ande72]
  - → fundamentalen Eigenschaften zur Charakterisierung von Sicherheitsarchitekturen
- Begriffe des Referenzmonitorprinzips kennen wir schon:
  - Abgrenzung passiver Ressourcen (in Form einzelner Objekte, z.B. Dateien)
  - von Subjekten (aktiven Elementen, z.B. laufenden Programmen, Prozessen) durch Betriebssystem

Referenzmonitorprinzip

- Idee:
  - → sämtliche Autorisierungsentscheidungen durch einen zentralen (abstrakten) Mechanismus = Referenzmonitor
  - Bewertet jeden Zugriffsversuch eines Subjekts auf Objekt durch Anwendung einer Sicherheitspolitik (security policy)
    - → vgl. SELinux
  - somit: Architekturbeschreibung, wie Zugriffe auf Ressourcen (z.B. Dateien) auf solche Zugriffe, die Sicherheitspolitik erlaubt, eingeschränkt werden
- Autorisierungsentscheidungen
  - basieren auf sicherheitsrelevanten Eigenschaften jedes Subjekts und jedes Objekts
  - einige Beispiele kennen wir schon:
    - Nutzname, Unix-Gruppe
    - Prozess-ID, INode-Nummer
    - SELinux-Typ
- Architekturkomponenten in a nutshell:

Definierende Eigenschaften: Referenzmonitor ist eine Architekturkomponenten, die

- (RM 1) bei sämtlichen Subjekt/Objekt-Interaktionen involviert sind
  - → Unumgehbarkeit (total mediation)
- (RM 2) geschützt sind vor unautorisierter Manipulation
  - → Manipulationssicherheit (tamperproofness)
- (RM 3) hinreichend klein und wohlstrukturiert sind, um formalen Analysemethoden zugänglich zu sein
  - → Verifizierbarkeit (verifyability)

Referenzmonitor in Betriebssystemen

Nahezu alle Betriebssysteme implementieren irgendeine Form eines Referenzmonitors [Jaeg11] und können über Begriffe, wie

- Subjekte
- Objekte
- Regeln einer Sicherheitspolitik charakterisiert sowie auf
- Unumgehbarkeit
- Manipulationssicherheit
- Verifizierbarkeit ihrer Sicherheitsarchitektur hin untersucht werden

Beispiel: Standard- Linux

- Subjekte (generell Prozesse)
  - haben reale (und effektive) Nutzer-Identifikatoren (UIDs)
- Objekte (verschiedene Systemressourcen, genutzt für Speicherung, Kommunikation: Dateien, Directories, Sockets, SharedMemory usw.)
  - haben ACLs („rwxrw----")
- Regeln der Sicherheitspolitik, die durch den Referenzmonitor (hier Kernel) unterstützt werden
  - hart codiert, starr
- Sicherheitsattribute, die durch diese Regeln zur Prüfung genutzt werden (z.B. Zugriffsmodi)
  - Objekten zugeordnet
  - modifizierbar

Man beurteile die Politikimplementierung in dieser Architektur bzgl.:

- Unumgehbarkeit
- Manipulationssicherheit
- Verifizierbarkeit

Referenzmonitorimplementierung: Flask (Flask - Architekturmodell)

SELinux-Architektur: Security Server

- Security Server: Laufzeitumgebung für Politik in Schutzdomäne des Kerns
- Objektmanager: implementiert in allen BS-Dienstenmittels,, Linux Security Module Framework ”
  - jedes Subsystemvon SELinux , das zuständig für
    1. Erzeugung neuer Objekte
    2. Zugriff auf existierende Objekte
  - Beispiele:
    1. Prozess-Verwaltung (behandelte Objekte: hauptsächlich Prozesse)
    2. Dateisystem (behandelte Objekte: hauptsächlich Dateien)
    3. Networking/Socket-Subsystem (behandelte Objekte: [verschiedene Typen von] Sockets)
    4. u.a.

SELinux-Architektur: Objektklassen

- Objektmanager zur Verwaltung verschiedener Objektklassen
- spiegeln Diversität und Komplexität von Linux BS-Abtraktionen wider:
  - Dateisysteme: file, dir, fd, filesystem, ...
  - Netzwerk: netif, socket, tcp_socket, udp_socket, ...
  - IPC: msgq, sem, shm, ...
  - Sonstige: process, system, ...
  - ...

Dateisystem als Objektmanager

- Durch Analyse von Linux - Dateisystem und zugehöriger API wurden zu überwachenden Objektklassen identifiziert:
  - ergibt sich unmittelbar aus Linux-API:
    - Dateien
    - Verzeichnisse
    - Pipes
  - feingranularere Objektklassen für durch Dateien repräsentierte Objekte (Unix-Prinzip: ,,everythingisa file”!):
    - reguläre Dateien
    - symbolische Links
    - zeichenorientierte Geräte
    - blockorientierte Geräte
    - FIFOs
    - Unix-Domain Sockets (lokale Sockets)
- Permissions (Zugriffsrechte)
- für jede Objektklasse: Menge an Permissions definiert, um Zugriffe auf Objekte dieser Klasse zu kontrollieren
- Permissions: abgeleitet aus Dienstleistungen, die Linux-Dateisystem anbietet
- → Objektklassen gruppieren verschiedene Arten von Zugriffsoperationen auf verschiende Arten von Objekten
- z.B. Permissions für alle ,,Datei”-Objektklassen (Auswahl ...): read, write, append, create, execute, unlink
- für ,,Verzeichnis”-Objektklasse: add_name, remove_name, reparant, search, rmdir

Trusted Computing Base (TCB)

Begriff zur Bewertung von Referenzmonitorarchitekturen: TCB (Trusted Computing Base)

- = die Hard-und Softwarefunktionen eines IT-Systems, die notwendig und hinreichend sind, um alle Sicherheitsregeln durchzusetzen.
- besteht üblicherweise aus
  1. Laufzeitumgebung der Hardware(nicht E/A-Geräte)
  2. verschiedenen Komponenten des Betriebssystem-Kernels
  3. Benutzerprogrammen mit sicherheitsrelevanten Rechten (bei Standard-UNIX/Linux-Systemen: diejenigen mit root-Rechten)
- Betriebssystemfunktionen, die Teil der TCB sein müssen, beinhalten Teile
  - des Prozessmanagements
  - des Speichermanagements
  - des Dateimanagements
  - des E/A-Managements
  - alle Referenzmonitorfunktionen

Echtzeitfähigkeit

Motivation

Echtzeitbegriff: Was ist ein Echtzeitsystem?

> Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness. (The Oxford DictionaryofComputing)
>
> A real-time system is any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period. [Young 1982]
>
> A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment. [Randall et.al. 1995]

Spektrum von Echtzeitsystemen:

1. Regelungssysteme: z.B. eingebettete Systeme (exakter: Steuerungs-, Regelungs-u. Überwachungssysteme = ,,SRÜ”-Systeme)
2. Endanwender-Rechnersysteme: z.B. Multimediasysteme
3. Lebewesen: Menschen, Tiere

Beispiel Regelungssystem: ,,Fly-by-Wire”-Fluglage-Regelungssystem (Schema)

1. Flugzeugbewegung
2. Sensoren + Einstellmöglichkeiten des Piloten
3. Echtzeit-Datenverarbeitung (durch Echtzeit-Rechnersystem)
4. Aktoren setzen Berechnung um
5. Einstellung von Regelflächen
6. Aerodynamik und Flug Mechanik führt zu Flugzeugbewegung (1.)

Beispiel Überwachungssysteme

- Luftraumüberwachung:
  - Ortsfeste Radarstation
  - Mobile Radarstation
  - Tieffliegеr-Erfassungsradar
  - Flugplatzradar Netzfunkstellen
  - Zentrale
- Umweltüberwachung: Stickstoffdioxidkonzentration über Europa
- Vorgeburtliche Gesundheitsüberwachung: Herzschlagsüberwachungssystem für Mutter und Kind

Beispiel Multimediasystem

- zeitabhängige Datenwiedergabe
- Bildwiedergabe bei Mediendatenströmen
- Durchführung der Schritte durch Multimedia-Task binnen $t_{i+1} - t_i$
- Frist für Rendering in Multimedia-Tasks: festgelegt durch periodische Bildrate (24˜48 fps → 1/24 ... 1/48 s)
- → Berücksichtigung bei Scheduling, Interruptbehandlung, Speicherverwaltung, ... erforderlich!

Zwischenfazit [Buttazzo97]

- Murphy‘s General Law: If something can go wrong, it will got wrong.
- Murphy‘s Constant: Damage to an object is proportional to its value.
- Johnson‘s First Law: If a system stops working, it will do it at the worst possible time.
- Sodd‘sSecond Law: Sooner or later, the worst possible combination of circumstances will happen.

Realisierung von Echtzeiteigenschaften: komplex und fragil!

Terminologie

bevor wir uns über Echtzeit-Betriebssystemen unterhalten:

1. Wie ist die Eigenschaft Echtzeit definiert?
2. Was sind (rechnerbasierte) Echtzeitsysteme?
3. Wie können Echtzeitanwendungen beschrieben werden?
4. Welche grundsätzlichen Typen von Echtzeitprozessen gibt es/wodurch werden diese charakterisiert?

Antwortzeit:

- Alle Definitionen -die zitierten u. andere - betrachten eine ,,responsetime” (Antwortzeit, Reaktionszeit) als das Zeitintervall, das ein System braucht, um (irgend)eine Ausgabe als Reaktion auf (irgend)eine Eingabe zu erzeugen.

Frist

- Bei Echtzeitsystemen ist genau dieses Δt kritisch, d.h. je nach Art des Systems darf dieses auf keinen Fall zu groß werden.
- Genauer spezifizierbar wird dies durch Einführung einer Frist (deadline, due time) d, die angibt bis zu welchem Zeitpunkt spätestmöglich die Reaktion erfolgt sein muss, bzw. wie groß das Intervall Δt maximal sein darf.

Echtzeitfähigkeit und Korrektheit

- Wird genau dieses maximale Zeitintervall in die Spezifikation eines Systems einbezogen, bedeutet dies, dass ein Echtzeitsystem nur dann korrekt arbeitet, wenn seine Reaktion bis zur spezifizierten Frist erfolgt.
- Die Frist trennt also korrektes von inkorrektem Verhalten des Systems.

Harte und weiche Echtzeitsysteme

- Praktische Anwendungen erfordern oft Unterscheidung in harte und weiche Echtzeitsysteme:
  - hartes Echtzeitsystem: keine Frist darf jemals überschritten werden (sonst: katastrophale Konsequenzen)
  - weiches Echtzeitsystem: maßvolles (im spezifizierten Maß) Überschreiten von Fristen tolerierbar

Charakteristika von Echtzeit-Prozessen

- reale Echtzeitanwendungen beinhalten periodische oder aperiodische Prozesse (oder Mischung aus beiden)
- typische Unterscheidung:
  - Periodische Prozesse
    - zeitgesteuert (typisch: periodische Sensorauswertung)
    - oft: kritische Aktivitäten → harte Fristen
  - Aperiodische Prozesse
    - ereignisgesteuert
    - Abhängig von Anwendung: harte oder weiche Fristen, ggf. sogar Nicht-Echtzeit

Periodische Prozesse

- bei Echtzeit-Anwendungen: häufigster Fall
- typisch für:
  1. periodische Analyse von Sensor-Daten (z.B. Umweltüberwachung)
  2. Aktionsplanung (z.B. automatisierte Montage)
  3. Erzeugung oder Verarbeitung einzelner Dateneinheiten eines multimedialen Datenstroms
  4. ...
- Prozessaktivierung
  - ereignisgesteuert oder zeitgesteuert
  - Prozesse, die Eingangsdaten verarbeiten: meist ereignisgesteuert, z.B. wenn neues Datenpaket eingetroffen
  - Prozesse, die Ausgangsdaten erzeugen: meist zeitgesteuert, z.B. Ansteuerung von Roboteraktoren

Periodische Prozesse

- Fristen:
  - hart oder weich (anwendungsabhängig)
    - innerhalb einer Anwendung sind sowohl Prozesse mit harten oder weichen Fristen möglich
    - Frist: spätestens am Ende der aktuellen Periode, möglich auch frühere Frist
- Modellierung:
  - unendliche Folge identischer Aktivierungen: Instanzen, aktiviert mit konstanter Rate (Periode)
- Aufgaben des Betriebssystems:
  - WennalleSpezifikationeneingehaltenwerden-muss Betriebssystem garantieren, dass
    1. zeitgesteuerte periodische Prozesse: mit ihrer spezifizierten Rate aktiviert werden und ihre Frist einhalten können
    2. ereignisgesteuerte periodische Prozesse: ihre Frist einhalten können

Aperiodische Prozesse

- typisch für
  - unregelmäßig auftretende Ereignisse, z.B.:
    - Überfahren der Spurgrenzen, Unterschreiten des Sicherheitsabstands → Reaktion des Fahrassistenzsystems
    - Nutzereingaben in Multimediasystemen (→ Spielkonsole)
- Prozessaktivierung
  - ereignisgesteuert
- Fristen
  - oft weich(aber anwendungsabhängig)
- Aufgabendes Betriebssystems
  - bei Einhaltung der Prozessspezifikationen muss Betriebssystem auch hier für Einhaltung der Fristen sorgen
- Modellierung
  - bestehen ebenfalls aus (maximal unendlicher) Folge identischer Aktivierungen (Instanzen); aber: Aktivierungszeitpunkte nicht regelmäßig (möglich: nur genau eine Aktivierung)

Parameter von Echtzeit-Prozessen

-
- a_i: Ankunftszeitpunkt (arrival time); auch r ... request time/release time
  - Zeitpunkt, zu dem ein Prozess ablauffähig wird
- s_i: Startzeitpunkt (start time)
  - Zeitpunkt, zu dem ein Prozess mit der Ausführung beginnt
- f_i: Beendigungszeitpunkt (finishing time)
  - Zeitpunkt, an dem ein Prozess seine Ausführung beendet
- d_i: Frist (deadline, due time)
  - Zeitpunkt, zu dem ein Prozess seine Ausführung spätestens beenden sollte
- C_i: Bearbeitungszeit(bedarf) (computation time)
  - Zeitquantum, das Prozessor zur vollständigen Bearbeitung der aktuellen Instanz benötigt (Unterbrechungen nicht eingerechnet)
-
- L_i: Unpünktlichkeit (lateness): $L_i= f_i - d_i$
  - Zeitbetrag, um den ein Prozess früher oder später als seine Frist beendet wird (wenn Prozess vor seiner Frist beendet, hat L_i negativen Wert)
- E_i: Verspätung (exceeding time, tardiness): $E_i= max(0, L_i)$
  - Zeitbetrag, den ein Prozess noch nach seiner Frist aktiv ist
-
- X_i: Spielraum (Laxity, Slacktime): $X_i = d_i - a_i - C_i$
  - maximales Zeitquantum, um das Ausführung eines Prozesses verzögert werden kann, damit dieser noch bis zu seiner Frist beendet werden kann ($f_i=d_i$)
- außerdem:
  - criticality: Parameter zur Beschreibung der Konsequenzen einer Fristüberschreitung (typischerweise ,,hart" oder ,,weich")
  - V_i ...Wert (value): Parameter zum Ausdruck der relativen Wichtigkeit eines Prozesses bezogen auf andere Prozesse der gleichen Anwendung

Echtzeitfähige Betriebssysteme

- Hauptfragestellungen
  1. Was muss BS zu tun, um Echtzeitprozesse zu ermöglichen? Welche Teilprobleme müssen beachtet werden?
  2. Welche Mechanismen müssen hierfür anders als bei nicht-echtzeitfähigen Betriebssystemen implementiert werden, und wie?
- Grundlegender Gedanke
  - Abgeleitet aus den Aufgaben eines Betriebssystems sind folgende Fragestellungenvon Interesse:
  1. Wie müssen die Ressourcen verwaltet werden? (→ CPU, Speicher, E/A, ...)
  2. Sind neue Abstraktionen, Paradigmen (Herangehensweisen) und entsprechende Komponenten erforderlich (oder günstig)?
- Prozess-Metainformationen
  1. Frist
  2. Periodendauer
  3. abgeleitet davon: Spielraum, Unpünktlichkeit, Verspätung, ...
  4. im Zusammenhang damit: Prioritätsumkehr, Überlast
- Ressourcen-Management
  - Wie müssen Ressourcen verwaltet werden, damit Fristen eingehalten werden können?

Wir betrachten i.F.

1. Algorithmen, die Rechnersysteme echtzeitfähig machen -einschließlich des Betriebssystems:
   - grundlegende Algorithmen zum Echtzeitscheduling
   - Besonderheiten der Interruptbehandlung
   - Besonderheiten der Speicherverwaltung
2. Probleme, die behandelt werden müssen, um Echtzeitfähigkeit nicht zu be- oder verhindern:
   - Prioritätsumkehr
   - Überlast
   - Kommunikation-und Synchronisationsprobleme

Echtzeitscheduling

- Scheduling:
  - Schedulingvon Prozessen/Threads als wichtigster Einflussfaktor auf Zeitverhalten des Gesamtsystems
- Echtzeit-Scheduling:
  - benötigt: Scheduling-Algorithmen, die Scheduling unter Berücksichtigung der (unterschiedlichen) Fristen der Prozesse durchführen können
- Fundamentale Algorithmen:
  - wichtigste Strategien:
  1. Ratenmonotones Scheduling (RM)
  2. Earliest Deadline First (EDF)
  - beide schon 1973 von Liu & Layland ausführlich diskutiert [Liu&Layland73]

Annahmen der Scheduling-Strategien

- A1: Alle Instanzen eines periodischen Prozesses t_i treten regelmäßig und mit konstanter Rate auf (= werden aktiviert). Das Zeitintervall T_i zwischen zwei aufeinanderfolgenden Aktivierungen heißt Periode des Prozesses.
- A2: Alle Instanzen eines periodischen Prozesses t_i haben den gleichen Worst-Case-Rechenzeitbedarf C_i.
- A3: Alle Instanzen eines periodischen Prozesses t_i haben die gleiche relative Frist D_i, welche gleich der Periodendauer T_i ist.
- A4: Alle Prozessesind kausal unabhängig voneinander (d.h. keine Vorrang- und Betriebsmittel-Restriktionen)
- A5: Kein Prozess kann sich selbst suspendieren, z.B. bei E/A-Operationen.
- A6: Alle Prozesse werden mit ihrer Aktivierung sofort rechenbereit (release time = arrival time).
- A7: Jeglicher Betriebssystem-Overhead (Kontextwechsel, Scheduler-Rechenzeit) wird vernachlässigt.

A5-7 sind weitere Annahmen des Scheduling Modells
Ratenmonotones Scheduling (RM)

- Voraussetzung:
  - periodisches Bereitwerden der Prozesse/Threads, d.h. periodische Prozesse bzw. Threads
- Strategie RM:
  - Prozess (Thread) mit höchster Ankunftsrate bekommt höchste statische Priorität (Kriterium: Wie oft pro Zeiteinheit wird Prozess bereit?)
  - Scheduling-Zeitpunkt: nur einmal zu Beginn (bzw. wenn neuer periodischer Prozess auftritt)
  - präemptiver Algorithmus

-
  - Zuteilung eines Prozessors nach RM
  - t_1, t_2: Anforderungen von Prozessorzeit durch zwei periodische Prozesse
  - darunter: Prozessorzuteilung nach RM
- Optimalität von RM
  - Unter allen Verfahren mit festen (statischen)Prioritäten ist RM optimaler Algorithmus in dem Sinne, dass kein anderes Verfahren dieser Klasse eine Prozessmenge einplanen kann, die nicht auch von RM geplant werden kann. [Liu&Layland73]
- Prozessor-Auslastungsfaktor
  - Bei gegebener Menge von n periodischen Prozessen gilt: $U=\sum_{i=1}^n \frac{C_i}{T_i}$
  - mit $\frac{C_i}{T_i}$ Anteil an Prozessorzeit für jeden periodischen Prozess t_i
  - und U Summe der Prozessorzeit zur Ausführung der gesamten Prozessmenge (,,utilization factor")
- Prozessorlast
  - U ist folglich Maß für die durch Prozessmenge verursachte Last am Prozessor → Auslastungsfaktor
- Planbarkeitsanalyse einer Prozessmenge
  - im allgemeinen Fall kann RM einen Prozessor nicht zu 100% auslasten
  - von besonderem Interesse: kleinste obere Grenze des Auslastungsfaktors U_{lub} (lub: ,,least upper bound")
- Beispiel für $n=2$
  - Obere Grenze des Prozessor-Auslastungsfaktors für zwei periodische Prozesse als Funktion des Verhältnisses ihrer Perioden.
  - (Abb. nach [Buttazzo97] Bild 4.7, S. 90)
- Obere Auslastungsgrenze bei RM
  - nach [Buttazzo97] (S. 89-91) erhält man bei n Prozessen für RM: $U_{lub}=n(2^{\frac{1}{n}}-1)$
  - für $n\rightarrow\infty$ konvergiert U_{lub} zu $ln\ 2 \approx 0,6931...$
  - Wird genannter Wert nicht überschritten, sind beliebige Prozessmengen planbar.
  - (Herleitung siehe [Buttazzo97] , Kap. 4.3.3)

Earliest Deadline First (EDF)

- Voraussetzung:
  - kann sowohl periodische als auch aperiodische Prozesse planen
- Optimalität:
  - EDF in Klasse der Schedulingverfahren mit dynamischen Prioritäten: optimaler Algorithmus [Liu&Layland73]
- Strategie EDF:
  - Zu jedem Zeitpunkt erhält Prozess mit frühester Frist höchste dynamische Priorität
  - Scheduling-Zeitpunkt: Bereitwerden eines (beliebigen) Prozesses
  - präemptiver Algorithmus (keine Verdrängung bei gleichen Prioritäten)
- Beispiel
  - Zuteilung eines Prozessors nach EDF
  - t_1, t_2: Anforderungen nach Prozessorzeit durch zwei periodische Prozesse
  - darunter: Prozessorzuteilung nach EDF
- Planbarkeitsanalyse:
  - Mit den Regeln $A1 ... A7$ ergibt sich für die obere Schranke des Prozessorauslastungsfaktors: $U_{lub}= 1\rightarrow$ Auslastung bis 100% möglich!
  - Eine Menge periodischer Prozesse ist demnach mit EDF planbar genau dann wenn: $U=\sum_{i=1}^n \frac{C_i}{T_i}\leq 1$ (Prozessor natürlich nicht mehr als 100% auslastbar)
- Beweis: Obere Auslastungsgrenze bei EDF
  - Behauptung: Jede Menge von n periodischen Tasks ist mit EDF planbar ↔: $U=\sum_{i=1}^n \frac{C_i}{T_i}\leq 1$
  - ←: $U>1$ übersteigt die verfügbare Prozessorzeit; folglich kann niemals eine Prozessmenge mit dieser (oder höherer) Gesamtauslastung planbar sein.
  - →: Beweis durch Widerspruch. Annahme: $U\leq 1$ und die Prozessmenge ist nicht planbar. Dies führt zu einem Schedule mit Fristverletzung zu einem Zeitpunkt t_2, z.B.:
  - Beobachtungen an diesem Schedule:
    - $exists$ ein längstes, kontinuierliches Rechenintervall $[t_1,t_2]$, in welchem nur Prozessinstanzen mit Fristen $\leq t_2$ rechnen
    - die Gesamtrechenzeit C_{bad} aller Prozesse in $[t_1,t_2]$ muss die verfügbare Prozessorzeit übersteigen: $C_{bad} > t_2-t_1$ (sonst: keine Fristverletzung an t_2)
    - Anwesenheit in $[t_1,t_2]$ leitet sich davon ab, ob (genauer: wie oft) die Periode eines Prozesses in t_2-t_1 passt: t_i in $[t_1,t_2]\Leftrightarrow\lfloor\frac{t_2-t_1}{T_i}\rfloor >0$
    - Damit ist C_{bad} die Summe der Rechenzeiten aller Prozessinstanzen, die garantiert in $[t_1,t_2]$ sind, mithin: $C_{bad}=\sum_{i=1}^n \lfloor\frac{t_2-t_1}{T_i}\rfloor C_i$
    - Im Beispiel: $t_1... t_3$ in $[t_1,t_2]$, folglich: $C_{bad}= 2 C_1 + 1 C_2 + 1 C_3$
    - Zu zeigen: Beobachtung $C_{bad}> t_2-t_1$ widerspricht Annahme $U\leq 1$.
    - Es gilt $\sum_{i=1}^n \lfloor\frac{t_2-t_1}{T_i}\rfloor C_i\leq\sum_{i=1}^n\frac{t_2-t_1}{T_i}C_i$ wegen Abrundung.
    - Mit $U=\sum_{i=1}^n\frac{C_i}{T_i}$ folgt daraus $C_{bad}\leq(t_2-t_1)U$
    - $C_{bad}>t_2-t_1$ entspricht also $(t_2-t_1)U>t_2-t_1$ und somit $U>1$. Widerspruch zur Annahme!

Vergleich: EDF vs. RM

Zuteilung eines Prozessors nach EDF (dynamisch) bzw. RM (statisch) t_1,t_2: Anforderungen nach Prozessorzeit durch zwei periodische Prozesse darunter: Prozessorzuteilung nach EDF bzw. RM

- gut erkennbar: deutliche Unterschiede bei Scheduling mit statischem (RM) vs. dynamischem Algorithmus (EDF).

Vergleich: Anzahl Prozesswechsel

- Häufigkeit von Prozesswechseln im Beispiel:
  - RM: 16
  - EDF: 12
- Ursache: dynamische Prioritätenvergabe führt dazu, dass Instanz II von t_2 die gleiche Priorität wie Instanz A von t_1 hat (usw.) → keine unnötige Verdrängung

Vergleich: 100% Prozessorauslastung

- EDF: erzeugt auch bei Prozessorauslastung bis 100% (immer) korrekte Schedules
- RM: kann das im allgemeinen Fall nicht
- Bedeutung von 100% Prozessorauslastung in der Praxis: Überwiegend müssen Systeme mit harten Echtzeitanforderungen auch weiche Echtzeit- sowie Nicht-Echtzeit-Prozesse unterstützen. Daher: Belegungslücken am Prozessor für die letzteren beiden nutzbar.

Vergleich: Implementierung

- RM
  - statisch: jeweils eine Warteschlange pro Priorität:
  - Einfügen und Entfernen von Tasks: $O(1)$
- EDF
  - dynamisch: balancierter Binärbaum zur Sortierung nach Prioritäten:
  - Einfügen und Entfernen von Tasks: $O(log\ n)$

Scheduling in Multimedia-Anwendungen

- Konkretisierung des Betrachtungswinkels
  - RM und EDF wurden entwickelt insbesondere für Echtzeit-Regelsysteme → ohne Berücksichtigung von Multimediasystemen
  - Multimediasysteme → andere Probleme, schwächere Annahmen: spezialisierte Scheduling-Algorithmen
  - gehen meist auch von EDF und/oder RM als Grundlage aus
- Betrachteter Algorithmus:
  - Beispielfür spezialisierten Scheduling-Algorithmus:
    - RC-Algorithmus - entwickelt an University of Texas
    - Anpassung von EDF an Charakteristika von Multimedia-Anwendungen

Prozesstypen in Multimedia-Anwendungen

1. Echte Multimedia-Prozesse
   - periodische Prozesse: weiche Fristen
     1. pünktliche periodische Prozesse mit konstantem Prozessorzeitbedarf C für jede Instanz (unkomprimierte Audio- und Videodaten)
     2. pünktliche periodische Prozesse mit unterschiedlichem C einzelner Instanzen (komprimierte Audio- und Videodaten)
     3. unpünktliche periodische Prozesse:
        - verspätete Prozesse
        - verfrühte Prozesse
   - aperiodische-Prozesse aus Multimedia-Anwendungen: weiche Fristen
2. Prozesse nebenläufiger Nicht-Multimedia-Anwendungen
   - interaktive Prozesse : keine Fristen , aber: keine zu langen Antwortzeiten Ansatz (z.B.): maximal tolerierbare Verzögerung
   - Hintergrund-Prozesse : zeitunkritisch, keine Fristen, aber : dürfen nicht verhungern

Multimediaanwendungen sind ein typisches Beispiel für mögliche Abweichungen der Lastpezifikation (T_i,C_i) eines Echtzeitprozesses! Problem: Abweichungen von Lastspezifikation

- gibt Prozessor nicht frei
- verspätete periodische Prozesse

RC Algorithmus

- Ziel
  - spezifikationstreue Prozesse nicht bestrafen durch Fristüberschreitung aufgrund abweichender Prozesse
- Idee
  - grundsätzlich: Schedulingnach frühester Fristaufsteigend (= EDF) → für eine vollständig spezifikationstreue Prozessmenge verhält sich RC wie reines EDF
  - Frist einer Instanz wird dynamisch angepasst:basierend auf derjenigen Periode, in der sie eigentlich sein sollte lt. Spezifikation der Prozessornutzung (U_i, hier: ,,Rate"): $U_i=\frac{C_i}{T_i}$
  - Bsp.: $U_i =\frac{20}{40}=\frac{1}{2}$ (t_B hat spezifizierte Aktivitätsrate von $0,5$ pro Periode)

RC Algorithmus: Strategie

- Variablen
  - a_i: Ankunftszeit der zuletzt bereitgewordenen Instanz von t_i
  - t_i^{virt}: virtuelle Zeit in aktueller Periode, die t_i bereits verbraucht hat
  - c_i^{virt}: Netto-Rechenzeit, die t_i in aktueller Periode bereits verbraucht hat
  - d_i: dynamische Frist von t_i, nach der sich dessen Priorität berechnet (EDF)
- Strategie
  - für eine bereite (lauffähige) Instanz von t_i: adaptiere dynamisch d_i basierend auf t_i^{virt}
  - für eine bereit gewordene (neu angekommene oder zuvor blockierte) Instanzvon t_i: aktualisiere t_i^{virt} auf akt. Systemzeit $(t)\rightarrow$ etwaiger "Zeitkredit" verfällt

RC Algorithmus: Berechnung von t_i^{virt}

Beispiel: Situation bei $t=20ms$
Da t_B aber noch weiteren Rechenbedarf hat: Situation bei $t=30 ms$

RC Algorithmus: Adaptionsfunktion

Für Prozess ti zu jedem Scheduling-Zeitpunkt:

RC Algorithmus: Scheduling

Zeitpunkte, zu denen der Scheduler aktiv wird:

1. aktuell laufender Prozess t_i blockiert:
   - $RC(t_i)$
2. Prozesse $t_i...._j$ werden bereit:
   - $for\ x\in[i,j]: RC(t_x)$
3. periodischer ,,clock tick" (SchedulingInterrupt):
   - t_i := aktuell ausgeführter Prozess
   - $RC(t_i)$

anschließendes Scheduling (präemptiv) = EDF:

Umgang mit abweichenden Prozessen unter RC

Resultat

Garantie: Prozesse, die sich entsprechend ihrer Spezifikation verhalten, erhalten bis zum Ende jeder spezifizierten Periode ihren spezifizierten Anteil an Prozessorzeit.

Auswirkung auf verschiedene Prozesstypen:

- ,,pünktliche" Prozesse: Einhaltung der Frist in jeder Periode garantiert (unabhängig von Verhalten anderer Prozesse)
- ,,verspätete" Prozesse: nur aktuelle Periode betrachtet, Nachholen ,,ausgelassener Perioden" nicht möglich
- ,,gierige" Prozesse: Prozessorentzug, sobald andere lauffähige Prozesse frühere Fristen aufweisen
- nicht-periodische Hintergrundprozesse: pro ,,Periode" wird spezifizierte Prozessorrate garantiert (z.B. kleine Raten bei großen ,,Periodendauern" wählen.)

Umgang mit gemischten Prozessmengen

- Hintergrund-Scheduling:
  - Prinzip:
  - rechenbereite Prozesse auf 2 Warteschlangen aufgeteilt (einfache Variante eines Mehr-Ebenen-Scheduling)
  - Warteschlange 1:
    - alle periodischen Prozesse
    - mit höchster Priorität mittels RM oder EDF bedient
  - Warteschlange 2:
    - alle aperiodischen Prozesse
    - nur bedient, wenn keine wartenden Prozesse in Warteschlange 1

Hintergrund-Scheduling: Vor- und Nachteile

- Hauptvorteil:
  - einfache Implementierung
- Nachteile:
  - Antwortzeit **aperiodischer Prozesse** kann zu lang werden (insbesondere bei hoher aperiodischer Last) → Verhungern möglich!
  - geeignet nur für relativ zeitunkritische aperiodische Prozesse
- Beispiel: Hintergrund-Scheduling mit RM

Optimierung: Server-Prozess

- Scheduling mit Server-Prozessen:
  - Prinzip: periodisch aktivierter Prozess benutzt zur Ausführung aperiodischer Prozessoranforderungen
  - Beschreibung Server-Prozess: durch Parameter äquivalent zu wirklichem periodischen Prozess:
    - Periodendauer T_S
    - ,,Prozessorzeitbedarf" C_S; jetzt Kapazitätdes Server-Prozesses
  - Arbeitsweise Server-Prozess:
    - geplant mit gleichem Scheduling-Algorithmus wie periodische Prozesse
    - zum Aktivierungszeitpunkt vorliegende aperiodische Anforderungen bedient bis zur Kapazität des Servers
    - keine aperiodischen Anforderungen: Server suspendiert sich bis Beginn der nächsten Periode (Schedule wird ohne ihn weitergeführt → Prozessorzeit für periodische Prozesse)
    - Kapazitätin jeder Server-Periode neu "aufgeladen"

Beispiel: Server-Prozess mit RM

Optimierung: Slack-Stealing

- Prinzip: Es existiert passiver Prozess ,,slack stealer" (kein periodischer Server)
- versucht so viel Zeit wie möglich für aperiodische Anforderungen zu sammeln
- realisiert durch ,,slackstealing"(= Spielraum-Stehlen) bei periodischen Prozessen
- letztere auf Zeit-Achse so weit nach hinten geschoben, dass Frist und Beendigungszeitpunkt zusammenfallen
- Sinnvoll, da normalerweise Beenden periodischer Prozesse vor ihrer Frist keinerlei Vorteile bringt
- Resultat: Verbesserung der Antwortzeiten für aperiodische Anforderungen
-

Prioritätsumkehr

Mechanismen zur Synchronisation und Koordination sind häufige Ursachen für kausale Abhängigkeiten zwischen Prozessen!

Problem

- Prinzip *kritischer Abschnitt* (Grundlagen BS):
  - Sperrmechanismen stellen wechselseitigen Ausschluss bei der Benutzung exklusiver Betriebsmittel durch nebenläufige Prozesse sicher
  - Benutzung von exklusiven sowie nichtentziehbaren Betriebsmitteln: kritischer Abschnitt
  - Folge: Wenn ein Prozess einen kritischen Abschnitt betreten hat, darf er aus diesem nicht verdrängt werden (durch anderen Prozess, der dasselbe Betriebsmittel nutzen will)
- Konflikt: kritische Abschnitte vs. Echtzeit-Prioritäten
  - Falls ein weiterer Prozess mit höherer Priorität ablauffähig wird und im gleichen kritischen Abschnitt arbeiten will, muss er warten bis niederpriorisierter Prozess kritischen Abschnitt verlassen hat
  - (zeitweise) Prioritätsumkehr möglich! d.h. aus einer (Teil-) Menge von Prozessen muss derjenige mit höchster Priorität auf solche mit niedrigerer Priorität warten

Ursache der Prioritätsumkehr

-
- Prioritätsumkehr bei Blockierung an nichtentziehbarem, exklusivem Betriebsmittel
- → unvermeidlich

Folgen der Prioritätsumkehr

- Kritisch bei zusätzlichen Prozessen mittlerer Priorität
- Lösung: Priority Inheritance Protocol (PIP)

Lösung: Prioritätsvererbung

- ![Abb. nach [Buttazzo97] , Bild 7.6, S.188]
- ePrio ... effektive Priorität

Überlast

- Definition: kritische Situation - bei der die benötigte Menge an Prozessorzeit die Kapazität des vorhandenen Prozessors übersteigt $(U>1)$
  - Folge: nicht alle Prozesse können Fristen einhalten
- Hauptrisiko: kritische Prozesse können Fristen nicht einhalten → Gefährdung funktionaler und anderer nichtfkt. Eigenschaften (→ harte Fristen!)
- Stichwort: ,,graceful degradation" (,,würdevolle" Verschlechterung) statt unkontrollierbarer Situation → Wahrung von Determinismus

Wichtigkeit eines Prozesses

- Minimallösung: (lebenswichtig für Echtzeit-System)
  - Unterscheidung zwischen Zeitbeschränkungen (Fristen) und tatsächlicher Wichtigkeit eines Prozesses für System

- Allgemein gilt:
  - Wichtigkeit eines Prozesses ist unabhängig von seiner Periodendauer und irgendwelchen Fristen
  - z.B. kann ein Prozess trotz späterer Frist viel wichtiger als anderer mit früherer Frist sein.
  - Beispiel: Bei chemischem Prozess könnte Temperaturauswertung jede 10s wichtiger sein als Aktualisierung graphischer Darstellung an Nutzerkonsole jeweils nach 5s

Umgang mit Überlast: alltägliche Analogien

1. Weglassen weniger wichtiger Aktionen
   - ohne Frühstück aus dem Haus...
   - kein Zähneputzen ...
   - Wichtung vom Problem bzw. Aktivitätsträgern (hier: Personen) abhängig!
2. Verkürzen von Aktivitäten
   - Katzenwäsche...
3. Kombinieren
   - kein Frühstück + Katzenwäsche + ungekämmt

Wichtung von Prozessen Behandlung:

- zusätzlicher Parameter V (Wert) für jeden Prozess/Thread einer Anwendung
- spezifiziert relative Wichtigkeit eines Prozesses (od. Thread) im Verhältnis zu anderen Prozessen (Threads) der gleichen Anwendung
- bei Scheduling: V stellt zusätzliche Randbedingung (primär: Priorität aufgrund von Frist, sekundär: Wichtigkeit)

Obligatorischer und optionaler Prozessanteil

- Aufteilung der Gesamtberechnung (C_{ges}) eines Prozesses in zwei Phasen
- einfache Möglichkeit der Nutzung des Konzepts des anpassbaren Prozessorzeitbedarfs
- Prinzip:
  - Bearbeitungszeitbedarf eines Prozesses zerlegt in
    1. obligatorischer Teil (Pflichtteil, C_{ob}): muss unbedingt u. immer ausgeführt werden → liefert bedingt akzeptables Ergebnis
    2. optionaler Teil (C_{opt}): nur bei ausreichender Prozessorkapazität ausgeführt → verbessert durch obligatorischen Teil erzieltes Ergebnis
  - Prinzip in unterschiedlicher Weise verfeinerbar
-

Echtzeit-Interruptbehandlung

1. Fristüberschreitung durch ungeeignete Interruptbearbeitung
2. Lösung für Echtzeitsysteme ohne Fristüberschreitung
   - Interrupt wird zunächst nur registriert (deterministischer Zeitaufwand)
   - tatsächliche Bearbeitung der Interruptroutine muss durch Scheduler eingeplant werden → Pop-up Thread

Echtzeit-Speicherverwaltung

- Prinzip:
  - Hauptanliegen: auch hier Fristen einhalten
  - wie bei Interrupt-Bearbeitung und Prioritätsumkehr: unkontrollierbare Verzögerungen der Prozessbearbeitung (= zeitlicher Nichtdeterminismus) vermeiden!
- Ressourcenzuordnung, deswegen:
  1. keine Ressourcen-Zuordnung ,,on-demand" (d.h. in dem Moment, wo sie benötigt werden) sondern ,,Pre-Allokation" (= Vorab-Zuordnung)
  2. keine dynamische Ressourcenzuordnung (z.B. Hauptspeicher), sondern Zuordnung maximal benötigter Menge bei Pre-Allokation (→ BS mit ausschließlich statischer Hauptspeicherallokation: TinyOS)

Hauptspeicherverwaltung

- bei Anwendung existierender Paging-Systeme
  - durch unkontrolliertes Ein-/Auslagern ,,zeitkritischer" Seiten (-inhalte): unkontrollierbare Zeitverzögerungen möglich!
  - Technik hier: ,,Festnageln" von Seiten im Speicher (Pinning, Memory Locking)

Sekundärspeicherverwaltung

- Beispiel 1: FCFS Festplattenscheduling
  - Anforderungsreihenfolge = 98, 183, 37, 122, 14, 124, 65, 67
  - Zuletzt gelesener Block: 53
- Beispiel 2: EDF Festplattenscheduling
  - Anforderungsreihenfolge $t_1 = 98, 37, 124, 65$
  - Anforderungsreihenfolge $t_2 = 183, 122, 14, 67$
  - Zuletzt gelesener Block: 53 | | a_i | d_i | | ----- | ----- | ----- | | t_1 | 0 | 3 | | t_2 | 0 | 9 |
- Primärziel: Wahrung der Echtzeitgarantien
  - naheliegend: EA-Schedulingnach Fristen → EDF (wie Prozessor)
  - für Zugriffsreihenfolge auf Datenblöcke: lediglich deren Fristen maßgebend (weitere Regeln existieren nicht!)
- Resultat bei HDDs:
  - ineffiziente Bewegungen der Lese-/Schreibköpfe -ähnlich FCFS
  - nichtdeterministische Positionierzeiten
  - geringer Durchsatz
- Fazit:
  - Echtzeit-Festplattenscheduling → Kompromiss zwischen Zeitbeschränkungen und Effizienz
- bekannte Lösungen:
  1. Modifikation von EDF
  2. Kombination von EDF mit anderen Zugriffsstrategien

→ realisierte Strategien:

1. SCAN-EDF (SCAN: Kopfbewegung nur in eine Richtung bis Mitte-/Randzylinder; EDF über alle angefragten Blöcke *in dieser Richtung*)
2. Group Sweeping_ (SCAN mit nach Fristen gruppenweiser Bedienung)
3. Mischstrategien

- Vereinfachung:
  - o.g. Algorithmen i.d.R. zylinderorientiert → berücksichtigen bei Optimierung nur Positionierzeiten (Grund: Positionierzeit meist >> Latenzzeit)

Kommunikation und Synchronisation

- zeitlichen Nichtdeterminismus vermeiden:
  1. Interprozess-Kommunikation
  - Minimierung blockierender Kommunikationsoperationen
  - indirekte Kommunikation → CAB zum Geschwindigkeitsausgleich
  - keine FIFO-Ordnungen (nach Fristen priorisieren)
  - CAB ... Cyclic Asynchronous Buffer:
  2. Synchronisation
  - keine FIFO-Ordnungen, z.B. bei Semaphor-Warteschlangen (vgl. o.)

Cyclic Asynchronous Buffer (CAB) Kommunikation zwischen 1 Sender und n Empfängern:

- nach erstem Schreibzugriff: garantiert niemals undefinierte Wartezeiten durch Blockierung von Sender/Empfänger
- Lesen/Überschreiben in zyklischer Reihenfolge:
- Implementierung:
  - MRW: Most-Recently-Written; Zeiger auf jüngstes, durch Sender vollständig geschriebenes Element
  - LRW: Least-Recently-Written; Zeiger auf ältestes durch Sender geschriebenes Element
  - Garantien:
    - sowohl *MRW* als auch *LRW* können ausschließlich durch Sender manipuliert werden → keine inkonsistenten Zeiger durch konkurrierende Schreibzugriffe!
    - sowohl *MRW* als auch *LRW* zeigen niemals auf ein Element, das gerade geschrieben wird → keine inkonsistenten Inhalte durch konkurrierende Schreib-/Lesezugriffe!
  - Regeln für Sender:
    - muss **nach** jedem Schreiben *MRW* auf geschriebenes Element setzen
    - muss **bevor** *LRW* geschrieben wird *LRW* inkrementieren
  - Regel für Empfänger: muss immer nach Lesen von *MRW* als nächstes *LRW* anstelle des Listennachbarn lesen
- Sender-Regeln:
  - anschaulich, ohne aktiven Empfänger
- Empfänger-Regel:
  - anschaulich, ohne aktiven Sender

Sonderfall 1: Empfänger schneller als Sender

- nach Zugriff auf MRW muss auf Lesesequenz bei LRW fortgesetzt werden → transparenter Umgang mit nicht-vollem Puffer
- Abschwächung der Ordnungsgarantien:Empfänger weiß nur, dass Aktualität der Daten zwischen LRW und MRW liegt
- Empfänger (nach min. einem geschriebenen Element) niemals durch leeren Puffer blockiert
-

Sonderfall 2: Sender schneller als Empfänger

- Schreiben in Puffer grundsätzlich in Reihenfolge der Elemente → keine blockierenden Puffergrenzen → niemals Blockierung des Senders
- keine Vollständigkeitsgarantien:Empfänger kann nicht sicher sein, eine temporal stetige Sequenz zu lesen
- → Szenarien, in denen Empfänger sowieso nur an aktuellsten Daten interessiert (z.B. Sensorwerte)
-

Konkurrierende Zugriffe:

- ... sind durch Empfänger immer unschädlich (da lesend)

- ... müssen vom Sender nach Inkrementieren von LRW nicht-blockierend erkannt werden (klassisches Semaphormodell ungeeignet)
- schnellerer Sender überspringtein gesperrtes Element durch erneutes Inkrementieren von LRW , muss MRW trotzdem nachziehen
-

Architekturen und Beispiel-Betriebssysteme

- Architekturprinzipien:
  - müssen Echtzeitmechanismen unterstützen; ermöglicht entsprechende Strategien zur Entwicklungs-oder Laufzeit (CPU-Scheduler, EA-Scheduler, IPC ...)
  - müssen funktional geringe Komplexität aufweisen → theoretische und praktische Beherrschung von Nichtdeterminismus
    - Theoretisch: Modellierung und Analyse (vgl. Annahmen für Scheduling-Planbarkeitsanalyse)
    - Praktisch: Implementierung (vgl. RC-Scheduler, Prioritätsvererbung)
- Konsequenzen:
  - Architekturen für komplementäre NFE:
    - Sparsamkeit → hardwarespezifische Kernelimplementierung
    - Adaptivität → μKernel, Exokernel
  - zu vermeiden:
    - starke Hardwareabstraktion → Virtualisierungsarchitekturen
    - Kommunikation und Synchronisationskosten → verteilte BS
    - Hardwareunabhängigkeit und Portabilität → vgl. Mach

Auswahl: Beispiel-Betriebssysteme

- wir kennen schon:
  - funktional kleine Kernelimplementierung: TinyOS
  - hardwarespezifischer μKernel: L4-Abkömmlinge
  - Mischung aus beidem: RIOT
  - Kommerziell bedeutender μKernel: QNX Neutrino
- weitere Vertreter:
  - hardwarespezifische Makrokernel: VRTX, VxWorks
  - μKernel: DRYOS, DROPS
  - ,,Exokernel" ...?

VRTX (Versatile Real-Time Executive)

- Entwickler:
  - Hunter & Ready
- Eckdaten:
  - Makrokernel
  - war erstes kommerzielles Echtzeitbetriebssystem für eingebettete Systeme
  - heutige Bedeutung eher historisch
  - Nachfolger (1993 bis heute): Nucleus RTOS (Siemens)
- Anwendung:
  - Eingebettete Systeme in Automobilen(Brems-und ABS-Controller)
  - Mobiltelefone
  - Geldautomaten
- Einsatzgebiete
  - spektakulär: im Hubble-Weltraumteleskop

VxWorks

- Entwickler:
  - Wind River Systems (USA)
- Eckdaten:
  - modularer Makrokernel
  - Konkurrenzprodukt zu VRTX
  - Erfolgsfaktor: POSIX-konforme API
  - ähnlich QNX: ,,skalierbarer" Kernel,zuschneidbarauf Anwendungsdomäne (→ Adaptivitätsansatz)
- Anwendung:
  - eingebettete Systeme:
  - industrielle Robotersteuerung
  - Luft-und Raumfahrt
  - Unterhaltungselektronik
- Einsatzgebiete
  - Deep-Impact-Mission zur Untersuchung des Kometen Temple 1
  - NASA Mars Rover
  - SpaceX Dragon

DRYOS®

- Entwickler: Canon Inc.
- Eckdaten:
  - Mikrokernel(Größe: 16 kB)
  - Echtzeit-Middleware (Gerätetreiber → Objektive)
  - Anwendungen: AE-und AF-Steuerung/-Automatik, GUI, Bildbearbeitung, RAW-Konverter, ...
  - POSIX-kompatible Prozessverwaltung
-

DROPS (Dresden Real-Time Operating System)

- Entwickler: TU Dresden, Lehrstuhl Betriebssysteme
- Eckdaten: Multi-Server-Architektur auf Basis eines L4-Mikrokerns
-

Adaptivität

Motivation

- als unmittelbar geforderte NFE:
  - eingebettete Systeme
  - Systeme in garstiger Umwelt (Meeresgrund, Arktis, Weltraum, ...)
  - Unterstützung von Cloud-Computing-Anwendungen
  - Unterstützung von Legacy-Anwendungen
- Beobachtung: genau diese Anwendungsdomänen fordern typischerweise auch andere wesentliche NFE (s.bisherige Vorlesung ...)
- → Adaptivität als komplementäre NFE zur Förderung von
  - Robustheit: funktionale Adaptivitätdes BS reduziert Kernelkomplexität (→ kleiner, nicht adaptiver μKernel)
  - Sicherheit: wie Robustheit:TCB-Größe → Verifizierbarkeit, außerdem: adaptive Reaktion auf Bedrohungen
  - Echtzeitfähigkeit: adaptive Scheduling-Strategie (vgl. RC), adapt. Überlastbehandlung, adapt. Interruptbehandlungs-und Pinning-Strategien
  - Performanz: Last-und Hardwareadaptivität
  - Erweiterbarkeit: adaptive BS liefern oft hinreichende Voraussetzungen der einfachen Erweiterbarkeit von Abstraktionen, Schnittstellen, Hardware-Multiplexing-und -Schutzmechanismen (Flexibility)
  - Wartbarkeit: Anpassung des BS an Anwendungen, nicht umgekehrt
  - Sparsamkeit: Lastadaptivitätvon CPUs, adaptive Auswahl von Datenstrukturen und Kodierungsverfahren

Adaptivitätsbegriff

- Adaptability: ,,see Flexibility. " [Marciniak94]
- Flexibility:
  - ,,The ease with which a system or a component can be modified for use in applications or environments other than those for which it was specifically designed." (IEEE)
  - für uns: entspricht Erweiterbarkeit
- Adaptivität: (unsere Arbeitsdefinition)
  - Die Fähigkeit eines Systems, sich an ein breites Spektrum verschiedener Anforderungen anpassen zu lassen.
  - = ... so gebaut zu sein, dass ein breites Spektrum verschiedener nicht funktionaler Eigenschaften unterstützt wird.
  - letztere: komplementär zur allgemeinen NFE Adaptivität

Roadmap

- in diesem Kapitel: gleichzeitig Mechanismen und Architekturkonzepte
- Adaptivität jeweils anhand komplementärer Eigenschaften dargestellt:
  - Exokernel: { Adaptivität } ∪ { Performanz, Echtzeitfähigkeit,Wartbarkeit, Sparsamkeit }
  - Virtualisierung: { Adaptivität } ∪ { Wartbarkeit, Sicherheit, Robustheit }
  - Container: { Adaptivität } ∪ { Wartbarkeit, Portabilität, Sparsamkeit }
- Beispielsysteme:
  - Exokernel-Betriebssysteme: Aegis/ExOS, Nemesis, MirageOS
  - Virtualisierung: Vmware, VirtualBox, Xen
  - Containersoftware: Docker

Exokernelarchitektur

- Grundfunktion von Betriebssystemen
  - physische Hardware darstellen als abstrahierte Hardware mit komfortableren Schnittstellen
  - Schnittstelle zu Anwendungen (API) : bietet dabei exakt die gleichen Abstraktionen der Hardware für alle Anwendungen an, z.B.
    - **Prozesse:** gleiches Zustandsmodell, gleiches Threadmodell
    - **Dateien:** gleiche Namensraumabstraktion
    - **Adressräume:** gleiche Speicherverwaltung (VMM, Seitengröße, Paging)
    - **Interprozesskommunikation:** gleiche Mechanismen für alle Anwendungsprozesse
- Problem:
  - Implementierungsspielraumfür Anwendungen wird begrenzt:

  1. Vorteile domänenspezifischer Optimierungender Hardwarebenutzung können nicht ausgeschöpft werden → **Performanz, Sparsamkeit**
  2. die Implementierung existierender Abstraktionen kann bei veränderten Anforderungen nicht an Anwendungen angepasst werden → **Wartbarkeit**
  3. Hardwarespezifikationen, insbesondere des Zeitverhaltens (E/A, Netzwerk etc.), werden von Effekten des BS-Management überlagert → **Echtzeitfähigkeit**
- Idee von Exokernel-Architekturen:

Exokernelmechanismen

- Designprinzip von Exokernelmechanismen:
  - Trennung von Schutz und Abstraktion der Ressourcen
  - Ressourcen-Schutz und -Multiplexing: verbleibt beim Betriebssystemkernel(dem Exokernel)
  - Ressourcen-Abstraktion (und deren Management): zentrale Aufgabe der Library-Betriebssysteme
    - → autonome Management-Strategien durch in Anwendungen importierte Funktionalität
  - Resultat:
    1. systemweit(durch jeweiliges BS vorgegebene) starre Hardware-Abstraktionen vermieden
    2. anwendungsdomänenspezifische Abstraktionen sehr einfach realisierbar
    3. (Wieder-) Verwendung eigener und fremder Managementfunktionalität wesentlich erleichtert → komplementäre NFEn! (Performanz, EZ-Fähigkeit, Sparsamkeit, ...)
- Funktion des Exokernels:
  - Prinzip: definiert Low-level-Schnittstelle
    - ,,low-level" = so hardwarenah wie möglich, bspw. die logische Schnittstelle eines elektronischen Schaltkreises/ICs (→ Gerätetreiber $\subseteq$ Library-BS!)
    - Bsp.: der Exokernelmuss den Hauptspeicher schützen, aber nicht verstehen, wie dieser verwaltet wird → Adressierung ermöglichen ohne Informationen über Seiten, Segmente, Paging-Attribute, ...
  - Library-Betriebssysteme: implementieren darauf jeweils geeignete anwendungsnahe Abstraktionen
    - Bsp.: Adressraumsemantik, Seitentabellenlayout und -verwaltung, Paging-und Locking-Verfahren, ...
  - Anwendungsprogrammierer: wählen geeignete Library-Betriebssysteme bzw. schreiben ihre eigenen Exokernelmechanismen
- prinzipielle Exokernelmechanismen am Beispiel Aegis/ExOS [Engler+95]
  - Der Exokernel...
    - *implementiert:* Multiplexing der Hardware-Ressourcen
    - *exportiert:* geschützte Hardware-Ressourcen
- minimal: drei Arten von Mechanismen
  1. Secure Binding: erlaubt geschützte Verwendung von Hardware-Ressourcen durch Anwendungen, Behandlung von Ereignissen
  2. Visible ResourceRevocation: beteiligt Anwendungen am Entzug von Ressourcen mittels (kooperativen) Ressourcen-Entzugsprotokolls
  3. Abort-Protokoll: erlaubt ExokernelBeendigung von Ressourcenzuordnungen bei unkooperativen Applikationen

Secure Binding

- Schutzmechanismus, der Autorisierung (→ Library-BS)zur Benutzung einer Ressource von tatsächlicher Benutzung (→ Exokernel) trennt
- implementiert für den Exokernelerforderliches Zuordnungswissenvon (HW-)Ressource zu Mangement-Code (der im Library-BS implementiert ist)
- → "Binding" in Aegis implementiert als Unix-Hardlinkauf Metadatenstruktur zu einem Gerät im Kernelspeicher (,,remember: everythingisa file...")
- Zur Implementierung benötigt:
  - Hardware-Unterstützung zur effizienten Rechteprüfung (insbes. HW-Caching)
  - Software-Caching von Autorisierungsentscheidungen im Kernel (bei Nutzung durch verschiedene Library-BS)
  - Downloadingvon Applikationscode in Kernel zur effizienten Durchsetzung (quasi: User-Space-Implementierung von Systemaufrufcode)
- einfach ausgedrückt: ,,Secure Binding" erlaubt einem ExokernelSchutz von Ressourcen, ohne deren Semantik verstehen zu müssen.

Visible Resource Revocation

- monolithische Betriebssysteme: entziehen Ressourcen ,,unsichtbar" (invisible), d.h. transparent für Anwendungen
  - Vorteil: im allgemeinen geringere Latenzzeiten, einfacheres und komfortableres Programmiermodell
  - Nachteil: Anwendungen(hier: die eingebetteten Library-BS) erhalten keine Kenntnis über Entzug,bspw. aufgrund von Ressourcenknappheit etc.
  - → erforderliches Wissen für Management-Strategien!
- Exokernel-Betriebssysteme: entziehen(überwiegend) Ressourcen ,,sichtbar" → Dialog zwischen Exokernel und Library-BS
  - Vorteil: effizientes Management durch Library-BS möglich (z.B. Prozessor: nur tatsächlich benötigte Register werden bei Entzug gespeichert)
  - Nachteil : Performanz bei sehr häufigem Entzug, Verwaltungs-und Fehlerbehandlungsstrategien zwischen verschiedenen Library-BS müssen korrekt und untereinander kompatibelsein...
  - → Abort - Protokoll notwendig, falls dies nicht gegeben ist

Abort - Protokoll

- Ressourcenentzug bei unkooperativen Library-Betriebssystemen (Konflikt mit Anforderung durch andere Anwendung/deren Library-BS: Verweigerung der Rückgabe, zu späte Rückgabe, ...)
- notwendig aufgrund von Visible Ressource Revocation
- Dialog:
  - Exokernel: ,,Bitte Seitenrahmen x freigeben."
  - Library-BS: ,,..."
  - Exokernel: ,,Seitenrahmen x innerhalb von 50 μs freigeben!"
  - Library-BS: ,,..."
  - Exokernel: (führt Abort-Protokoll aus)
  - Library-BS: X (,,Abort" in diesem Bsp. = Anwendungsprozess terminieren)
- In der Praxis:
  - harte Echtzeit-Fristen (,, innerhalb von 50 μs") in den wenigsten Anwendungen berücksichtigt
    - → Abort = lediglich Widerruf aller Secure Bindings der jeweiligen Ressource für die unkooperativeAnwendung, nicht deren Terminierung (= unsichtbarerRessourcenentzug)
    - → anschließend: Informieren des entsprechenden Library-BS
  - ermöglicht sinnvolle Reaktion des Library-BS (in Library-BS wird ,,Repossession"-Exceptionausgelöst, so dass auf Entzug geeignet reagiert werden kann)
  - bei zustandsbehafteten Ressourcen (→ CPU): Exokernelkann diesen Zustand auf Hintergrundspeicher sichern → Management-Informationen zum Aufräumen durch Library-BS

Exokernelperformanz

- Was macht Exokern-Architekturen adaptiv(er)?
  - Abstraktionen und Mechanismen des Betriebssystems können den Erfordernissen der Anwendungen angepasst werden
  - (erwünschtes) Ergebnis: beträchtliche Performanzsteigerungen (vgl. komplementäre Ziel-NFE: Performanz, Echtzeitfähigkeit, Wartbarkeit, Sparsamkeit)

Performanzstudien

1. Aegis mit Library-BS ExOS (MIT: Dawson Engler, Frans Kaashoek)
2. Xok mit Library-BS ExOS (MIT)
3. Nemesis (Pegasus-Projekt, EU)
4. XOmB (U Pittsburgh)
5. ...

Aegis/ExOSals erweiterte Machbarkeitsstudie [Engler+95]

1. machbar: sehr effiziente Exokerne
   - Grundlage: begrenzte Anzahl einfacher Systemaufrufe (Größenordnung ~10) und Kernel-interne Primitiven (,,Pseudo-Maschinenanweisungen"), die enthalten sein müssen
2. machbar: sicheres Hardware-Multiplexing auf niedriger Abstraktionsebene (,,low-level") mit geringem Overhead
3. traditionelle Abstraktionen (VMM, IPC) auf Anwendungsebene effizient implementierbar → einfache Erweiterbarkeit, Spezialisierbarkeitbzw. Ersetzbarkeit dieser Abstraktionen
4. für Anwendungen: hochspezialisierte Implementierungen von Abstraktionen generierbar, die genau auf Funktionalität und Performanz-Anforderungen dieser Anwendung zugeschnitten
5. geschützte Kontrollflussübergabe: als IPC-Primitive im Aegis-Kernel, 7-mal schnellerals damals beste Implementierung (vgl. [Liedtke95], Kap. 3)
6. Ausnahmebehandlung bei Aegis: 5-mal schneller als bei damals bester Implementierung
7. durch Aegis möglich: Flexibilität von ExOS, die mit Mikrokernel-Systemen nicht erreichbar ist:
   - Bsp. VMM: auf Anwendungsebene implementiert, wo diese sehr einfach mit DSM-Systemen u. Garbage-Kollektoren verknüpfbar
8. Aegis erlaubt Anwendungen Konstruktion effizienter IPC-Primitiven ($\Delta\mu$Kernel: nicht vertrauenswürdige Anwendungen können keinerlei spezialisierte IPC-Primitiven nutzen, geschweige denn selbst implementieren)

Xok/ExOS

- praktische Weiterentwicklung von Aegis: Xok
- für x86-Hardware implementiert
- Kernel-Aufgaben (wie gehabt): Multiplexing von Festplatte, Speicher, Netzwerkschnittstellen, ...
- Standard Library-BS (wie bei Aegis): ExOS
  - ,,Unix as a Library"
  - Plattform für unmodifizierte Unix-Anwendungen (csh, perl, gcc, telnet, ftp, ...)
- z.B. Library-BS zum Dateisystem-Management: C-FFS
  - hochperformant (im Vergleich mit Makrokernel-Dateisystem-Management)
  - Abstraktionen und Operationen auf Exokernel-Basis (u.a.): Inodes, Verzeichnisse, physische Dateirelokation(→ zusammenhängendes Lesen)
  - Secure Bindings für Metadaten-Modifikation
- Forschungsziele:
  - Aegis: Proof-of-Concept
  - XOK: Proof-of-Feasibility (Performanz)
-

Zwischenfazit: Exokernelarchitektur

- Ziele:

  - Performanz, Sparsamkeit: bei genauer Kenntnis der Hardware ermöglicht deren direkte BenutzungAnwendungsentwicklern Effizienzoptimierung
  - Wartbarkeit: Hardwareabstraktionen sollen flexibel an Anwendungsdomänen anpassbar sein, ohne das BS modifizieren/wechseln zu müssen
  - Echtzeitfähigkeit: Zeitverhaltendes Gesamtsystems durch direkte Steuerung der Hardware weitestgehend durch (Echtzeit-) Anwendungen kontrollierbar
- Idee:
  - User-Space:anwendungsspezifische Hardwareabstraktionen im User-Space implementiert
  - Kernel-Space:nur Multiplexing und Schutz der HW-Schnittstellen
  - in der Praxis: kooperativer Ressourcenentzug zwischen Kernel, Lib. OS
- Ergebnisse:
  - hochperformanteHardwarebenutzung durch spezialisierte Anwendungen
  - funktional kleiner Exokernel(→ Sparsamkeit, Korrektheit des Kernelcodes)
  - flexible Nutzung problemgerechterHW-Abstraktionen (readymade Lib. OS)
  - keine Isolation von Anwendungen (→ Parallelisierbarkeit: teuer und mit schwachen Garantien; → Robustheit und Sicherheit der Anwendungen: nicht umsetzbar)

Virtualisierung

- Ziele (zur Erinnerung):
  - Adaptivität
  - Wartbarkeit, Sicherheit, Robustheit
  - → auf gleicher Hardware mehrere unterschiedliche Betriebssysteme ausführbar machen
- Idee:

Ziele von Virtualisierung

- Adaptivität: (ähnlich wie bei Exokernen)
  - können viele unterschiedliche Betriebssysteme - mit jeweils unterschiedlichen Eigenschaften ausgeführt werden damit können: Gruppen von Anwendungen auf ähnliche Weise jeweils unterschiedliche Abstraktionen etc. zur Verfügung gestellt werden
- Wartbarkeit:
  - Anwendungen - die sonst nicht gemeinsam auf gleicher Maschine lauffähig - auf einer phyischenMaschine ausführbar
  - ökonomische Vorteile: Cloud-Computing, Wartbarkeit von Legacy-Anwendungen
- Sicherheit:
  - Isolation von Anwendungs-und Kernelcode durch getrennte Adressräume (wie z.B. bei Mikrokern-Architekturen)
  - somit möglich:
    1. Einschränkung der Fehlerausbreitung → angreifbare Schwachstellen
    2. Überwachung der Kommunikation zwischen Teilsystemen
  - darüber hinaus: Sandboxing (vollständig von logischer Ablaufumgebung isolierte Software, typischerweise Anwendungen → siehe z.B. Cloud-Computing)
- Robustheit:
  - siehe Sicherheit!

Architekturvarianten - drei unterschiedliche Prinzipien:

1. Typ-1 - Hypervisor (früher: VMM - ,,Virtual MachineMonitor")
2. Typ-2 - Hypervisor
3. Paravirtualisierung

Typ-1 - Hypervisor

-
- Idee des Typ- 1 - Hypervisors:
  - Kategorien traditioneller funktionaler Eigenschaften von BS:
    1. Multiplexing & Schutz der Hardware (ermöglicht Multiprozess-Betrieb)
    2. abstrahierte Maschine** mit ,,angenehmerer" Schnittstelle als die reine Hardware (z.B. Dateien, Sockets, Prozesse, ...)
- Typ- 1 - Hypervisor trennt beide Kategorien:
  - läuft wie ein Betriebssystem unmittelbar über der Hardware
  - bewirkt Multiplexing der Hardware, liefert aber keine erweiterte Maschine** an Anwendungsschicht → ,,Multi-Betriebssystem-Betrieb"
- Bietet mehrmals die unmittelbare Hardware-Schnittstelle an, wobei jede Instanz eine virtuelle Maschine jeweils mit den unveränderten Hardware-Eigenschaften darstellt (Kernel u. User Mode, Ein-/Ausgaben usw.).
- Ursprünge: Time-Sharing an Großrechnern
  - Standard-BS auf IBM-Großrechner System/360: OS/360
  - reines Stapelverarbeitungs-Betriebssystem (1960er Jahre)
  - Nutzer (insbes. Entwickler) strebten interaktive Arbeitsweise an eigenem Terminal an → timesharing (MIT, 1962: CTSS)
    - IBM zog nach: CP/CMS, später VM/370 → z/VM
    - CP: Control Program → Typ- 1 - Hypervisor
    - CMS: ConversationalMonitor System → Gast-BS
  - CP lief auf ,,blanker" Hardware (Begriff geprägt: ,,bare metal hypervisor")
    - lieferte Menge virtueller Kopiender System/360-Hardware an eigentliches Timesharing-System
    - je eines solche Kopie pro Nutzer → unterschiedliche BS lauffähig (da jede virtuelle Maschine exakte Kopie der Hardware)
    - in der Praxis: sehr leichtgewichtiges, schnelles Einzelnutzer-BS als Gast → CMS (heute wäre das wenig mehr als ein Terminal-Emulator...)
- heute: Forderungen nach Virtualisierung von Betriebssystemen
  - seit 1980er: universeller Einsatz des PC für Einzelplatz- und Serveranwendungen → veränderte Anforderungen an Virtualisierung
  - Wartbarkeit: vor allem ökonomische Gründe:
    1. Anwendungsentwicklung und -bereitstellung: verschiedene Anwendungen in Unternehmen, bisher auf verschiedenen Rechnern mit mehreren (oft verschiedenen) BS, auf einem Rechner entwickeln und betreiben (Lizenzkosten!)
    2. Administration: einfache Sicherung, Migration virtueller Maschinen
    3. Legacy-Software
  - später: Sicherheit, Robustheit → Cloud-Computing-Anwendungen
- ideal hierfür: Typ- 1 - Hypervisor
  - ✓Gast-BS angenehm wartbar
  - ✓Softwarekosten beherrschbar
  - ✓Anwendungen isolierbar

Hardware-Voraussetzungen

- Voraussetzungen zum Einsatz von Typ-1-HV
  - Ziel: Nutzung von Virtualisierung auf PC-Hardware
  - systematische Untersuchung der Virtualisierbarkeit von Prozessoren bereits 1974 durch Popek & Goldberg [Popek&Goldberg74]
  - Ergebnis:
    - Gast-BS (welches aus Sicht der CPU im User Mode - also unprivilegiert läuft) muss sicher sein können, dass privilegierte Instruktionen (Maschinencode im Kernel) ausgeführt werden
    - dies geht nur, wenn tatsächlich der HV diese Instruktionen ausführt!
    - dies geht nur, wenn CPU bei jeder solchen Instruktion im Nutzermodus Kontextwechsel zum HV ausführen, welcher Instruktion emuliert!
- virtualisierbare Prozessoren bis ca. 2006:
  - ✓IBM-Architekturen(bekannt: PowerPC, bis 2006 Apple-Standard)
  - ✗Intel x86-Architekturen (386, Pentium, teilweise Core i)

Privilegierte Instruktionen **ohne** Hypervisor

- kennen wir schon: Instruktion für Systemaufrufe

1. User Mode: Anwendung bereitet Befehl und Parameter vor
2. User Mode: Privilegierte Instruktion (syscall/Trap - Interrupt) → CPU veranlasst Kontext-und Privilegierungswechsel, Ziel: BS-Kernel
3. Kernel Mode: BS-Dispatcher (Einsprungpunkt für Kernel-Kontrollfluss) behandelt Befehl und Parameter, ruft weitere privilegierte Instruktionen auf (z.B. EA-Code)

-

Privilegierte Instruktionen mit Typ- 1 - Hypervisor(1)

- zum Vergleich: Instruktion für Systemaufrufe des Gast-BS

1. User Mode: Anwendung bereitet Befehl und Parameter vor
2. User Mode: Trap → Kontext-und Privilegierungswechsel, Ziel: Typ-1-HV
3. Kernel Mode: HV-Dispatcher ruft Dispatcher im Gast-BS auf
4. User Mode: BS-Dispatcher behandelt Befehl und Parameter, ruft weitere privilegierte Instruktionenauf (z.B. EA-Code) → Kontext-und Privilegierungswechsel, Ziel: Typ-1-HV
5. Kernel Mode: HV führt privilegierte Instruktionen anstelle des Gast-BS aus

-

Sensible und privilegierte Instruktionen: Beobachtungen an verschiedenen Maschinenbefehlssätzen: [Popek&Goldberg74]

- $\exists$ Menge an Maschinenbefehlen, die nur im Kernel Mode ausgeführt werden dürfen (Befehle zur Realisierung von E/A, Manipulation der MMU, ...)
  - → sensible Instruktionen
- $\exists$ Menge an Maschinenbefehlen, die Wechsel des Privilegierungsmodus auslösen (x86: Trap), wenn sie im User Mode ausgeführt werden
  - → privilegierte Instruktionen
- Prozessor ist virtualisierbarfalls (notw. Bed.): sensible Instruktionen $\subseteq$ privilegierte Instruktionen
- Folge: jeder Maschinenbefehl, der im Nutzermodus nicht erlaubt ist, muss einen Privilegierungswechsel auslösen (z.B. Trap generieren)
- kritische Instruktionen = sensible Instruktionen \ privilegierte Instruktionen
  - Befehle, welche diese Bedingung verletzen → Existenz im Befehlssatz führt zu nicht-virtualisierbarem Prozessor

- Beispiele für sensible Instruktionen bei Intel x86:
  - hlt: Befehlsabarbeitung bis zum nächsten Interrupt stoppen
  - invlpg: TLB-Eintrag für Seite invalidieren
  - lidt: IDT (interrupt descriptor table) neu laden
  - mov auf Steuerregistern
  - ...
- Beispiel: Privilegierte Prozessorinstruktionen
  - Bsp.: write - Systemaufruf
  - Anwendungsprogramm schreibt String in Puffer eines Ausgabegeräts ohne Nutzung der libc Standard-Bibliothek: `asm ( int $0x80" ); /* interrupt 80 (trap) */`
  - Interrupt-Instruktion veranlasst Prozessor zum Kontextwechsel: Kernelcode im privilegierten Modus ausführen

Vergleich: Privilegierte vs. sensible Instruktionen

-

Folgen für Virtualisierung

- privilegierte Instruktionen bei virtualisierbaren Prozessoren
- bei Ausführung einer privilegierten Instruktion in virtueller Maschine: immer Kontrollflussübergabe an im Kernel-Modus laufende Systemsoftware - hier Typ-1-HV
- HV kann (anhand des virtuellen Privilegierungsmodus) feststellen:
  1. ob sensible Anweisung durch Gast-BS
  2. oder durch Nutzerprogramm (Systemaufruf!) ausgelöst
- Folgen:
  1. privilegierte Instruktionen des Gast-Betriebssystems werden ausgeführt → ,,trap-and-emulate"
  2. Einsprung in Betriebssystem, hier also Einsprung in Gast-Betriebssystem → Upcall durch HV
- privilegierte Instruktionen bei nicht virtualisierbaren Prozessoren
  - solche Instruktionen typischerweise ignoriert!

Intel-Architektur ab 386

- dominant im PC-und Universalrechnersegment ab 1980er
- keine Unterstützung für Virtualisierung ...
- kritische Instruktionen im User Mode werden von CPU ignoriert
- außerdem: in Pentium-Familie konnte Kernel-Code explizit feststellen, ob er im Kernel- oder Nutzermodus läuft → Gast-BS trifft (implementierungsabhängig) evtl. fatal fehlerhafte Entscheidungen
- Diese Architekturprobleme (bekannt seit 1974) wurden 20 Jahre lang im Sinne von Rückwärtskompatibilität auf Nachfolgeprozessoren übertragen ...
  - erste virtualisierungsfähige Intel-Prozessorenfamilie (s. [Adams2006]): VT, VT-x® (2005)
  - dito für AMD: SVM, AMD-V® (auch 2005)

Forschungsarbeit 1990er Jahre

- verschiedene akademische Projekte zur Virtualisierung bisher nicht virtualisierbarer Prozessoren
- erstes und vermutlich bekanntestes: DISCO- Projekt der University of Stanford
- Resultat: letztlich VMware (heute kommerziell) und Typ-2-Hypervisors...

Typ-2-Hypervisor

Virtualisierung ohne Hardwareunterstützung:

- keine Möglichkeit, trap-and-emulate zu nutzen
- keine Möglichkeit, um
  1. korrekt (bei sensiblen Instruktionen im Gast-Kernel) den Privilegierungsmodus zu wechseln
  2. den korrekten Code im HV auszuführen

Übersetzungsstrategie in Software:

- vollständige Übersetzung des Maschinencodes, der in VM ausgeführt wird, in Maschinencode, der im HV ausgeführt wird
- praktische Forderung: HV sollte selbst abstrahierte HW-Schnittstelle zur Ausführung des (komplexen!) Übersetzungscodes zur Verfügung haben (z.B. Nutzung von Gerätetreibern)
- → Typ-2-HV als Kompromiss:
  - korrekte Ausführung von virtualisierter Software auf virtualisierter HW
  - beherrschbare Komplexität der Implementierung

aus Nutzersicht

- läuft als gewöhnlicher Nutzer-Prozess auf Host-Betriebssystem (z.B. Windows oder Linux)
- VMware bedienbarwie physischer Rechner (bspw. erwartet Bootmedium in virtueller Repräsentation eines physischen Laufwerks)
- persistente Daten des Gast-BS auf virtuellem Speichermedium (tatsächlich: Image-Datei aus Sicht des Host-Betriebssystems)

Mechanismus: Code-Inspektion

- Bei Ausführung eines Binärprogramms in der virtuellen Maschine (egal ob Bootloader, Gast-BS-Kernel, Anwendungsprogramm): zunächst inspiziert Typ-2-HV den Code nach Basisblöcken
  - Basisblock: Befehlsfolge, die mit privilegierten Befehlen oder solchen Befehlen abgeschlossen ist, die den Kontrollfluss ändern (sichtbar an Manipulation des Programm-Zählers eip), z.B. jmp, call, ret.
- Basisblöcke werden nach sensiblen Instruktionen abgesucht
- diese werden jeweils durchAufruf einer HV-Prozedur ersetzt, die jeweilige Instruktion behandelt
- gleiche Verfahrensweise mit letzter Instruktion eines Basis-Blocks

Mechanismus: Binary Translation (Binärcodeübersetzung)

- modifizierter Basisblock: wird innerhalbdes HVin Cachegespeichert und ausgeführt
- Basisblock ohne sensible Instruktionen: läuft unter Typ-2-HV exakt so schnell wie unmittelbar auf Hardware (weil er auch tatsächlich unmittelbar auf der Hardware läuft, nur eben im HV-Kontext)
- sensible Instruktionen: nach dargestellter Methode abgefangen und emuliert → dabei hilft jetzt das Host-BS (z.B. durch eigene Systemaufrufe, Gerätetreiberschnittstellen)

Mechanismus: Caching von Basisblöcken

- HV nutzt zwei parallel arbeitende Module (Host-BS-Threads!):
  - Translator: Code-Inspektion, Binary Translation
  - Dispatcher: Basisblock-Ausführung
- zusätzliche Datenstruktur: Basisblock-Cache
- Dispatcher: sucht Basisblock mit jeweils nächster auszuführender Befehlsadresse im Cache; falls miss → suspendieren (zugunsten Translator)
- Translator: schreibt Basisblöcke in Basisblock-Cache
- Annahme: irgendwann ist Großteil des Programms im Cache, dieses läuft dann mit nahezu Original-Geschwindigkeit (theoretisch)

Performanzmessungen

- zeigen gemischtes Bild: Typ2-HV keinesfalls so schlecht, wie einst erwartet wurde
- qualitativer Vergleich mit virtualisierbarer Hardware (Typ1-Hypervisor):
- ,,trap-and-emulate,,: erzeugt Vielzahl von Traps → Kontextwechsel zwischen jeweiliger VM und HV
- insbesondere bei Vielzahl an VMs sehr teuer: CPU-Caches, TLBs, Heuristiken zur spekulativen Ausführung werden verschmutzt
- wenn andererseits sensible Instruktionen durch Aufruf von VMware-Prozeduren innerhalb des ausführenden Programms ersetzt: keine Kontextwechsel-Overheads

Studie: (von Vmware) [Adams&Agesen06]

- last-und anwendungsabhängig kann Softwarelösung sogar Hardwarelösung übertreffen
- Folge: viele moderne Typ1-HV benutzen aus Performanzgründen ebenfalls Binary Translation

Paravirtualisierung

Funktionsprinzip

- ... unterscheidet sich prinzipiell von Typ-1/2-Hypervisor
- wesentlich: Quellcode des Gast-Betriebssystems modifiziert
- sensible Instruktionen: durch Hypervisor-Calls ersetzt
- Folge: Gast-Betriebssystem arbeitet jetzt vollständig wie Nutzerprogramm, welches Systemaufrufe zum Betriebssystem (hier dem Hypervisor) ausführt
- dazu:
  - Hypervisor: muss geeignetes Interface definieren (HV-Calls)
  - → Menge von Prozedur-Aufrufen zur Benutzung durch Gast-Betriebssystem
  - bilden eine HV-API als Schnittstelle für Gast-Betriebssysteme (nicht für Nutzerprogramme!)
- mehr dazu: Xen

Verwandtschaft mit Mikrokernel-Architekturen

- Geht man vom Typ-1-HV noch einen Schritt weiter ...
  - und entfernt alle sensiblen Instruktionen aus Gast-Betriebssystem ...
  - und ersetzt diese durch Hypervisor-Aufrufe, um Systemdienste wie E/A zu benutzen, ...
  - hat man praktisch den Hypervisor in Mikrokernel transformiert.
- ... und genau das wird auch schon gemacht: L^4Linux (TU Dresden)
  - Basis: stringente $L^4\mu$ Kernel-Implementierung (Typ-1-HV-artiger Funktionsumfang)
  - Anwendungslaufzeitumgebung: paravirtualisierter Linux-Kernel als Serverprozess
  - Ziele: Isolation (Sicherheit, Robustheit), Echtzeitfähigkeit durch direktere HW-Interaktion (vergleichbar Exokernel-Ziel)

Zwischenfazit Virtualisierung

- Ziele: Adaptivität komplementär zu...
  - Wartbarkeit : ökonomischer Betrieb von Cloud-und Legacy-Anwendungen ohne dedizierte Hardware
  - Sicherheit : sicherheitskritische Anwendungen können vollständig von nichtvertrauenswürdigen Anwendungen (und untereinander) isoliert werden
  - Robustheit : Fehler in VMs (= Anwendungsdomänen) können nicht andere VMs beeinträchtigen
- Idee: drei gängige Prinzipien:
  - Typ-1-HV: unmittelbares HW-Multiplexing, trap-and-emulate

  - Typ-2-HV: HW-Multiplexing auf Basis eines Host-OS, binarytranslation
  - Paravirtualisierung: Typ-1-HV für angepasstes Gast-OS, kein trap-and-emulate nötig → HV ähnelt μKern
- Ergebnisse:
  - ✓VMs mit individuell anpassbarer Laufzeitumgebung
  - ✓isolierteVMs
  - ✓kontrollierbare VM-Interaktion (untereinander und mit HW)
  - ✗keine hardwarespezifischen Optimierungen aus VM heraus möglich → Performanz, Echtzeitfähigkeit, Sparsamkeit!

Container

Ziele:

- Adaptivität , im Dienste von ...
- ... Wartbarkeit: einfachen Entwicklung, Installation, Rekonfiguration durch Kapselung von
  - Anwendungsprogrammen
  - * durch sie benutzte Bibliotheken
  - * Instanzen bestimmter BS-Ressourcen
- ... Portabilität: Betrieb von Anwendungen, die lediglich von einem bestimmten BS-Kernel abhängig sind (nämlich ein solcher, der Container unterstützt); insbesondere hinsichtlich:
  - Abhängigkeitskonflikten (Anwendungen und Bibliotheken)
  - fehlenden Abhängigkeiten (Anwendungen und Bibliotheken)
  - Versions-und Namenskonflikten
- ... Sparsamkeit: problemgerechtes ,,Packen,, von Anwendungen in Container → Reduktion an Overhead: selten (oder gar nicht) genutzter Code, Speicherbedarf, Hardware, ...

Idee:

- private Sichten (Container) bilden = private User-Space-Instanzen für verschiedene Anwendungsprogramme
- Kontrolle dieser Container i.S.v. Multiplexing, Unabhängigkeit und API: BS-Kernel
- somit keine Form der BS-Virtualisierung, eher: ,,User-Space-Virtualisierung,,

Anwendungsfälle für Container

- Anwendungsentwicklung:
  - konfliktfreies Entwickeln und Testen unterschiedlicher Software, für unterschiedliche Zielkonfigurationen BS-User-Space
- Anwendungsbetrieb und -administration:
  - Entschärfung von ,,dependency hell,,
  - einfache Migration, einfaches Backup von Anwendungen ohne den (bei Virtualisierungsimages als Ballast auftretenden) BS-Kernel
  - einfache Verteilung generischer Container für bestimmte Aufgaben
  - = Kombinationen von Anwendungen
- Anwendungsisolation? → Docker

Zwischenfazit: Container

- Ziele: Adaptivität komplementär zu...
  - Wartbarkeit : Vermeidung von Administrationskosten für Laufzeitumgebung von Anwendungen
  - Portabilität : Vereinfachung von Abhängigkeitsverwaltung
  - Sparsamkeit : Optimierung der Speicher-und Verwaltungskosten für Laufzeitumgebung von Anwendungen
- Idee:
  - unabhängige User-Space-Instanz für jeden einzelnen Container
  - Aufgaben des Kernels: Unterstützung der Containersoftware bei Multiplexing und Herstellung der Unabhängigkeitdieser Instanzen
- Ergebnisse:
  - ✓vereinfachte Anwendungsentwicklung
  - ✓vereinfachter Anwendungsbetrieb
  - ✗Infrastruktur nötig über (lokale) Containersoftware hinaus, um Containern zweckgerecht bereitzustellen und zu warten
  - ✗keine vollständige Isolationmöglich

Beispielsysteme (Auswahl)

- Virtualisierung: VMware, VirtualBox
- Paravirtualisierung: Xen
- Exokernel: Nemesis, MirageOS, RustyHermit
- Container: Docker, LupineLinux

Hypervisor

VMware

- "... ist Unternehmenin PaloAlto, Kalifornien (USA)
- gegründet 1998 von 5 Informatikern
- stellt verschiedene Virtualisierungs-Softwareprodukte her:
  1. VMware Workstation
     - war erstes Produkt von VMware (1999)
     - mehrere unabhängige Instanzen von x86- bzw. x86-64-Betriebssystemen auf einer Hardware betreibbar
  2. VMware Fusion: ähnliches Produkt für Intel Mac-Plattformen
  3. VMware Player: (eingestellte) Freeware für nichtkommerziellen Gebrauch
  4. VMware Server (eingestellte Freeware, ehem. GSX Server)
  5. VMware vSphere (ESXi)
     - Produkte 1 ... 3: für Desktop-Systeme
     - Produkte 4 ... 5: für Server-Systeme
     - Produkte 1 ... 4: Typ-2-Hypervisor
- bei VMware-Installation: spezielle vm- Treiber in Host-Betriebssystem eingefügt
- diese ermöglichen: direkten Hardware-Zugriff
- durch Laden der Treiber: entsteht ,,Virtualisierungsschicht" (VMware-Sprechweise)
-
  - Typ1- Hypervisor- Architektur
  - Anwendung nur bei VMware ESXi
  - Entsprechende Produkte in Vorbereitung

VirtualBox

- Virtualisierungs-Software für x86- bzw. x86-64-Betriebssysteme für Industrie und ,,Hausgebrauch" (ursprünglich: Innotek , dann Sun , jetzt Oracle)
- frei verfügbare professionelle Lösung, als Open Source Software unter GNU General Public License(GPL) version 2. ...
- (gegenwärtig) lauffähig auf Windows, Linux, Macintosh und Solaris Hosts
- unterstützt große Anzahl von Gast-Betriebssystemen: Windows (NT 4.0, 2000, XP, Server 2003, Vista, Windows 7), DOS/Windows 3.x, Linux (2.4 and 2.6), Solaris and OpenSolaris , OS/2 , and OpenBSD u.a.
- reiner Typ-2-Hypervisor

Paravirutalisierung: Xen

- entstanden als Forschungsprojekt der University of Cambridge (UK), dann XenSource Inc., danach Citrix, jetzt: Linux Foundation (,,self-governing")
- frei verfügbar als Open Source Software unter GNU General Public License (GPL)
- lauffähig auf Prozessoren der Typen x86, x86-64, PowerPC, ARM, MIPS
- unterstützt große Anzahl von Gast-Betriebssystemen: FreeBSD, GNU/Hurd/Mach, Linux, MINIX, NetBSD, Netware, OpenSolaris, OZONE, Plan 9
- ,,Built for the cloud before it was called cloud." (Russel Pavlicek, Citrix)
- bekannt für Paravirtualisierung
- unterstützt heute auch andere Virtualisierungs-Prinzipien

Xen : Architektur

- Gast-BSe laufen in Xen Domänen (,,dom_i", analog VM_i)
- es existiert genau eine, obligatorische, vertrauenswürdige Domäne: dom_0
- Aufgaben (Details umseitig):
  - Bereitstellen und Verwalten der virtualisierten Hardware für andere Domänen (Hypervisor-API, Scheduling-Politiken für Hardware-Multiplexing)
  - Hardwareverwaltung/-kommunikation für paravirtualisierte Gast-BSe (Gerätetreiber)
  - Interaktionskontrolle (Sicherheitspolitiken)
- dom_0 im Detail: ein separates, hochkritisch administriertes, vertrauenswürdiges BS mit eben solchen Anwendungen (bzw. Kernelmodulen) zur Verwaltung des gesamten virtualisierten Systems
  - es existieren hierfür spezialisierte Variantenvon Linux, BSD, GNU Hurd

Xen : Sicherheit

- Sicherheitsmechanismusin Xen: Xen Security Modules (XSM)
- illustriert, wie (Para-) Typ-1-Virtualisierung von BS die NFE Sicherheit unterstützt
- PDP: Teil des vertrauenswürdigen BS in dom_0, PEPs: XSMs im Hypervisor
- Beispiel: Zugriff auf Hardware
  - Sicherheitspolitik-Integration, Administration, Auswertung: dom_0
- Beispiel: Inter-Domänen-Kommunikation
  - Interaktionskontrolle (Aufgaben wie oben): dom_0
  - Beispiel: VisorFlow
  - selber XSM kontrolliert Kommunikation für zwei Domänen

Exokernel

Nemesis

- Betriebssystemaus EU-Verbundprojekt ,,Pegasus,, zur Realisierung eines verteilten multimediafähigen Systems (1. Version: 1994/95)
- Entwurfsprinzipien:
  1. Anwendungen: sollen Freiheit haben, Betriebsmittel in für sie geeignetster Weise zu nutzen (= Exokernel-Prinzip)
  2. Realisierung als sog. vertikal strukturiertes Betriebssystem:
     - weitaus meiste Betriebssystem-Funktionalität innerhalb der Anwendungen ausgeführt (= Exokernel-Prinzip)
     - Echtzeitanforderungen durch Multimedia → Vermeidung von Client-Server-Kommunikationsmodell wegen schlecht beherrschbarer zeitlicher Verzögerungen (neu)
-

MirageOS + Xen

- Spezialfall: Exokernel als paravirtualisiertes BS auf Xen
- Ziele : Wartbarkeit (Herkunft: Virtualisierungsarchitekturen ...)

  - ökonomischer HW-Einsatz
  - Unterstützung einfacher Anwendungsentwicklung
  - nicht explizit: Unterstützung von Legacy-Anwendungen!
- Idee: ,,Unikernel" → eine Anwendung, eine API, ein Kernel
- umfangreiche Dokumentation, Tutorials, ... → ausprobieren
- Unikernel - Idee
  - Architekturprinzip:
  - in MirageOS:
- Ergebnis: Kombination von Vorteilen zweier Welten
  - Virtualisierungs vorteile: Sicherheit, Robustheit (→ Xen - Prinzip genau einer vertrauenswürdigen, isolierten Domäne dom_0)
  - Exokernelvorteile: Wartbarkeit, Sparsamkeit
  - nicht: Exokernelvorteil der hardwarenahen Anwendungsentwicklung... (→ Performanz und Echzeitfähigkeit)

Container: Docker

- Idee: Container für einfache Wartbarkeit von Linux-Anwendungsprogrammen ...
  - ... entwickeln
  - ... testen
  - ... konfigurieren
  - ... portieren → Portabilität
- Besonderheit: Container können - unabhängig von ihrem Einsatzzweck - wie Software-Repositories benutzt, verwaltet, aktualisiert, verteilt ... werden
- Management von Containers: Docker Client → leichtgewichtiger Ansatz zur Nutzung der Wartbarkeitsvorteile von Virtualisierung
- Forsetzung unter der OCI (Open Container Initiative)
  - ,,Docker does a nice job [...] for a focused purpose, namely the lightweight packaging and deployment of applications." (Dirk Merkel, Linux Journal)
- Implementierung der Containertechnik basierend auf Linux-Kernelfunktionen:
  - Linux Containers (LXC): BS-Unterstützung für Containermanagement
  - cgroups: Accounting/Beschränkung der Ressourcenzuordnung
  - union mounting: Funktion zur logischen Reorganisation hierarchischer Dateisysteme
-

Performanz und Parallelität

Motivation

- Performanz: Wer hätte gern einen schnell(er)en Rechner...?
- Wer braucht schnelle Rechner:
  - Hochleistungsrechnen, HPC (,,high performancecomputing")
    - wissenschaftliches Rechnen(z.B. Modellsimulation natürlicher Prozesse, Radioteleskop-Datenverarbeitung)
    - Datenvisualisierung(z.B. Analysen großer Netzwerke)
    - Datenorganisation-und speicherung(z.B. Kundendatenverarbeitung zur Personalisierung von Werbeaktivitäten, Bürgerdatenverarbeitung zur Personalisierung von Geheimdienstaktivitäten)
  - nicht disjunkt dazu: kommerzielle Anwendungen
    - ,,Big Data": Dienstleistungen für Kunden, die o. g. Probleme auf gigantischen Eingabedatenmengen zu lösen haben (Software wie Apache Hadoop)
    - Wettervorhersage
  - anspruchsvolle Multimedia- Anwendungen
    - Animationsfilme
    - VR-Rendering

Performanzbegriff

- Performance: The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage. (IEEE)
- Performanz im engeren Sinne dieses Kapitels: Minimierung der für korrekte Funktion (= Lösung eines Berechnungsproblems) zur Verfügung stehenden Zeit.
- oder technischer: Maximierung der Anzahl pro Zeiteinheit abgeschlossener Berechnungen.

Roadmap

- Grundlegende Erkenntnis: Performanz geht nicht (mehr) ohne Parallelität → Hochleistungsrechnen = hochparalleles Rechnen
- daher in diesem Kapitel: Anforderungen hochparallelen Rechnens an ...
  - Hardware: Prozessorarchitekturen
  - Systemsoftware: Betriebssystemmechanismen
  - Anwendungssoftware: Parallelisierbarkeitvon Problemen
- BS-Architekturen anhand von Beispielsystemen:
  - Multikernel: Barrelfish
  - verteilte Betriebssysteme

Hardware-Voraussetzungen

- Entwicklungstendenzen der Rechnerhardware:
  - Multicore-Prozessoren: seit ca. 2006 (in größerem Umfang)
  - Warum neues Paradigma für Prozessoren? bei CPU-Taktfrequenz >> 4 GHz: z.Zt. physikalische Grenze, u.a. nicht mehr sinnvoll handhabbare Abwärme
  - Damit weiterhin:
    1. Anzahl der Kerne wächst nicht linear
    2. Taktfrequenz wächst asymptotisch, nimmt nur noch marginal zu

Performanz durch Parallelisierung ...

Folgerungen

1. weitere Performanz-Steigerung von Anwendungen: primär durch Parallelität (aggressiverer) Multi-Threaded-Anwendungen
2. erforderlich: Betriebssystem-Unterstützung → Scheduling, Sychronisation
3. weiterhin erforderlich: Formulierungsmöglichkeiten (Sprachen), Compiler, verteilte Algorithmen ... → hier nicht im Fokus

... auf Prozessorebene

Vorteile von Multicore-Prozessoren

1. möglich wird: **Parallelarbeit auf Chip-Ebene** → Vermeidung der Plagen paralleler verteilter Systeme
2. bei geeigneter Architektur: Erkenntnisse und Software aus Gebiet verteilter Systeme als Grundlage verwendbar
3. durch gemeinsame Caches (architekturabhängig): schnellere Kommunikation (speicherbasiert), billigere Migration von Aktivitäten kann möglich sein
4. höhere Energieeffizienz: mehr Rechenleistung pro Chipfläche, geringere elektrische Leistungsaufnahme → weniger Gesamtabwärme, z.T. einzelne Kerne abschaltbar (vgl. Sparsamkeit , mobile Geräte)
5. Baugröße: geringeres physisches Volumen

Nachteile von Multicore-Prozessoren

1. durch gemeinsam genutzte Caches und Busstrukturen: Engpässe (Bottlenecks) möglich
2. zur Vermeidung thermischer Zerstörungen: Lastausgleich zwingend erforderlich! (Ziel: ausgeglichene Lastverteilung auf einzelnen Kernen)
3. zum optimalen Einsatz zwingend erforderlich:
   1. Entwicklung Hardwarearchitektur
   2. zusätzlich: Entwicklung geeigneter Systemsoftware
   3. zusätzlich: Entwicklung geeigneter Anwendungssoftware

Multicore-Prozessoren

- Sprechweise in der Literatur gelegentlich unübersichtlich...
- daher: Terminologie und Abkürzungen:
  - MC ...multicore(processor)
  - CMP ...chip-level multiprocessing, hochintegrierte Bauweise für ,,MC"
  - SMC ...symmetric multicore → SMP ... symmetric multi-processing
  - AMC ...asymmetric (auch: heterogeneous) multicore → AMP ... asymmetric multi-processing
  - UP ...uni-processing , Synonym zu singlecore(SC) oder uniprocessor

Architekturen von Multicore-Prozessoren

- A. Netzwerkbasiertes Design
  - Prozessorkerne des Chips u. ihre lokalen Speicher (oder Caches): durch Netzwerkstruktur verbunden
  - damit: größte Ähnlichkeit zu traditionellen verteilten Systemen
  - Verwendung: bei Vielzahl von Prozessorkernen (Skalierbarkeit!)
  - Beispiel: Intel Teraflop-Forschungsprozessor Polaris (80 Kerne als 8x10-Gitter)
- B. Hierarchisches Design
  - mehrere Prozessor-Kerne teilen sich mehrere baumartig angeordnete Caches
  - meistens:
    - jeder Prozessorkern hat eigenen L1-Cache
    - L2-Cache, Zugriff auf (externen) Hauptspeicher u. Großteil der Busse aber geteilt
  - Verwendung: typischerweise Serverkonfigurationen
  - Beispiele:
    - IBM Power
    - Intel Core 2, Core i
    - Sun UltraSPARCT1 (Niagara)
- C. Pipeline-Design
  - Daten durch mehrere Prozessor-Kerne schrittweise verarbeitet
  - durch letzten Prozessor: Ablage im Speichersystem
  - Verwendung:
    - Graphikchips
    - (hochspezialisierte) Netzwerkprozessoren
  - Beispiele: Prozessoren X10 u. X11 von Xelerator zur Verarbeitung von Netzwerkpaketen in Hochleistungsroutern (X11: bis zu 800 Pipeline-Prozessorkerne)

Symmetrische u. asymmetrische Multicore-Prozessoren

- symmetrische Multicore-Prozessoren (SMC)
  - alle Kerne identisch, d.h. gleiche Architektur und gleiche Fähigkeiten
  - Beispiele:
    - Intel Core 2 Duo
    - Intel Core 2 Quad
    - ParallaxPropeller
- asymmetrische MC-Prozessoren (AMC)
- Multicore-Architektur, jedoch mit Kernen unterschiedlicher Architektur und/oder unterschiedlichen Fähigkeiten
- Beispiel: Kilocore:
  - 1 Allzweck-Prozessor (PowerPC)
  - 256 od. 1024 Datenverarbeitungsprozessoren

Superskalare Prozessoren

- Bekannt aus Rechnerarchitektur: Pipelining
  - parallele Abarbeitung von Teilen eines Maschinenbefehls in Pipeline-Stufen
  - ermöglicht durch verschiedene Funktionseinheiten eines Prozessors für verschiedene Stufen:
    - Control Unit (CU)
    - ArithmeticLogicUnit (ALU)
    - Float Point Unit (FPU)
    - Memory Management Unit (MMU)
    - Cache
  - sowie mehrere Pipeline-Register
- superskalare Prozessoren: solche, bei denen zur Bearbeitung einer Pipeling-Stufe erforderlichen Funktionseinheiten n-fach vorliegen
- Ziel:
  - Skalarprozessor (mit Pipelining): 1 Befehl pro Takt (vollständig) bearbeitet
  - Superskalarprozessor: bis zu n Befehle pro Taktbearbeitet
- Verbereitung heute: universell (bis hin zu allen Desktop-Prozessorfamilien)

Parallelisierung in Betriebssystemen

- Basis für alle Parallelarbeit aus BS-Sicht: Multithreading
- wir erinnern uns ...:
  - Kernel-Level-Threads (KLTs): BS implementiert Threads → Scheduler kann mehrere Threads nebenläufig planen → Parallelität möglich
  - User-Level-Threads (ULTs): Anwendung implementiert Threads → keine Parallelität möglich!
- grundlegend für echt paralleles Multithreading:
  - parallelisierungsfähige Hardware
  - kausal unabhängige Threads
  - passendes (und korrekt eingesetztes!) Programmiermodell, insbesondere Synchronisation!
  - → Programmierer + Compiler

Vorläufiges Fazit:

- BS-Abstraktionen müssen Parallelität unterstützen (Abstraktion nebenläufiger Aktivitäten: KLTs)
- BS muss Synchronisationsmechanismen implementieren

Synchronisations- und Sperrmechanismen

- Synchronisationsmechanismen zur Nutzung
  - ... durch Anwendungen → Teil der API
  - ... durch den Kernel (z.B. Implementierung Prozessmanagement, E/A, ...)
- Aufgabe: Verhinderung konkurrierender Zugriffe auf logische oder physische Ressourcen
  - Vermeidung von raceconditions
  - Herstellung einer korrekten Ordnung entsprechend Kommunikationssemantik (z.B. ,,Schreiben vor Lesen")
- (alt-) bekanntes Bsp.: Reader-Writer-Problem

Erinnerung: Reader-Writer-Problem

- Begriffe: (bekannt)
  - wechselseitiger Ausschluss (mutual exclusion)
  - kritischer Abschnitt (critical section)
- Synchronisationsprobleme:
  - Wie verhindern wir ein write in vollen Puffer?
  - Wie verhindern wir ein read aus leerem Puffer?
  - Wie verhindern wir, dass auf ein Element während des read durch ein gleichzeitiges write zugegriffen wird? (Oder umgekehrt?)

Sperrmechanismen (Locks)

- Wechselseitiger Ausschluss ...
  - ... ist in nebenläufigen Systemen zwingend erforderlich
  - ... ist in echt parallelen Systemen allgegenwärtig
  - ... skaliert äußerst unfreundlich mit Code-Komplexität → (monolithischer) Kernel-Code!
- Mechanismen in Betriebssystemen: Locks
- Arten von Locks am Beispiel Linux:
  - Big Kernel Lock (BKL)
    - historisch (1996-2011): lockkernel(); ... unlockkernel();
    - ineffizient durch massiv gestiegene Komplexität des Kernels
  - atomic-Operationen
  - Spinlocks
  - Semaphore (Spezialform: Reader/Writer Locks)

atomic*

- Bausteine der komplexeren Sperrmechanismen:
  - Granularität: einzelne Integer- (oder sogar Bit-) Operation
  - Performanz: mittels Assembler implementiert, nutzt Atomaritäts garantienden CPU (TSL - Anweisungen: ,,test-set-lock")
- Benutzung:
  - `atomic_*` Geschmacksrichtungen: read, set, add, sub, inc, dec u. a.
  - keine explizite Lock-Datenstruktur → Deadlocks durch Mehrfachsperrung syntaktisch unmöglich
  - definierte Länge des kritischen Abschnitts (genau diese eine Operation) → unnötiges Sperren sehr preiswert

Zusammenfassung

Funktionale und nichtfunktionale Eigenschaften

- Funktionale Eigenschaften: beschreiben, was ein (Software)-Produkt tun soll
- Nichtfunktionale Eigenschaften: beschreiben, wie funktionale Eigenschaften realisiert werden, also welche sonstigen Eigenschaftendas Produkt haben soll ... unterteilbar in:
  1. Laufzeiteigenschaften (zur Laufzeit sichtbar)
  2. Evolutionseigenschaften (beim Betrieb sichtbar: Erweiterung, Wartung, Test usw.)

Roadmap (... von Betriebssystemen)

- Sparsamkeit und Effizienz
- Robustheit und Verfügbarkeit
- Sicherheit
- Echtzeitfähigkeit
- Adaptivität
- Performanzund Parallelität

Sparsamkeit und Effizienz

- Sparsamkeit: Die Eigenschaft eines Systems, seine Funktion mit minimalem Ressourcenverbrauch auszuüben.
- Effizienz: Der Grad, zu welchem ein System oder eine seiner Komponenten seine Funktion mit minimalem Ressourcenverbrauch ausübt. → Ausnutzungsgrad begrenzter Ressourcen
- Die jeweils betrachtete(n) Ressource(n) muss /(müssen) dabei spezifiziert sein!
- sinnvolle Möglichkeiten bei Betriebssystemen:
  1. Sparsamer Umgang mit Energie , z.B. energieeffizientes Scheduling
  2. Sparsamer Umgang mit Speicherplatz (Speichereffizienz)
  3. Sparsamer Umgang mit Prozessorzeit
  4. ...

Sparsamkeit mit Energie

- Sparsamkeit mit Energie als heute extrem wichtigen Ressource, mit nochmals gesteigerter Bedeutung bei mobilen bzw. vollständig autonomen Geräten Maßnahmen:

1. Hardware-Ebene: momentan nicht oder nicht mit maximaler Leistung benötigte Ressourcen in energiesparenden Modus bringen: abschalten, Standby, Betrieb mit verringertem Energieverbrauch (abwägen gegen verminderte Leistung). (Geeignete Hardware wurde/wird ggf. erst entwickelt)
2. Software-Ebene: neue Komponenten entwickeln, die in der Lage sein müssen:
   - Bedingungenzu erkennen, unter denen ein energiesparender Modus möglich ist;
   - Steuerungs-Algorithmen für Hardwarebetrieb so zu gestalten, dass Hardware-Ressourcen möglichst lange in einem energiesparenden Modus betrieben werden.
   - Energie-Verwaltungsstrategien: energieeffizientes Scheduling zur Vermeidung von Unfairness und Prioritätsumkehr
   - Beispiele: energieeffizientes Magnetfestplatten-Prefetching, energiebewusstes RR-Scheduling

Sparsamkeit mit Speicherplatz

- Betrachtet: Sparsamkeit mit Speicherplatz mit besonderer Wichtigkeit für physisch beschränkte, eingebettete und autonome Geräte
- Maßnahmen Hauptspeicherauslastung:
  1. Algorithmus und Strategie z.B.:
     - Speicherplatz sparende Algorithmen zur Realisierung gleicher Strategien
  2. Speicherverwaltung von Betriebssystemen:
     - physische vs. virtuelle Speicherverwaltung
     - speichereffiziente Ressourcenverwaltung
     - Speicherbedarfdes Kernels
     - direkte Speicherverwaltungskosten
- Maßnahmen Hintergrundspeicherauslastung:
  1. Speicherbedarf des Betriebssystem-Images
  2. dynamische SharedLibraries
  3. VMM-Auslagerungsbereich
  4. Modularität und Adaptivität des Betriebssystem-Images
- Nicht betrachtet: Sparsamkeit mit Prozessorzeit → 99% Überschneidung mit NFE Performanz

Robustheit und Verfügbarkeit

- Robustheit: Zuverlässigkeit unter Anwesenheit externer Ausfälle
- fault, aktiviert → error, breitet sich aus → failure

Robustheit

- Erhöhung der Robustheit durch Isolation:
  - Maßnahmen zur Verhinderung der Fehlerausbreitung:
  1. Adressraumisolation: Mikrokernarchitekturen,
  2. kryptografische HW-Unterstützung: Intel SGX und
  3. Virtualisierungsarchitekturen
- Erhöhung der Robustheit durch Behandlung von Ausfällen: Micro-Reboots

Vorbedingung für Robustheit: Korrektheit

- Korrektheit: Eigenschaft eines Systems sich gemäß seiner Spezifikation zu verhalten (unter der Annahme, dass bei dieser keine Fehler gemacht wurden).
- Maßnahmen (nur angesprochen):

1. diverse Software-Tests:
   - können nur Fehler aufspüren, aber keine Fehlerfreiheit garantieren!
2. Verifizierung:
   - Durch umfangreichen mathematischen Apparat wird Korrektheit der Software bewiesen.
   - Aufgrund der Komplexität ist Größe verifizierbarer Systeme (bisher?) begrenzt.
   - Betriebssystem-Beispiel: verifizierter Mikrokern seL

Verfügbarkeit

- Verfügbarkeit: Der Anteil an Laufzeit eines Systems, in dem dieses seine spezifizierte Leistung erbringt.
- angesprochen: Hochverfügbare Systeme
- Maßnahmen zur Erhöhung der Verfügbarkeit:

1. Robustheitsmaßnahmen
2. Redundanz
3. Redundanz
4. Redundanz
5. Ausfallmanagement

Sicherheit

- Sicherheit (IT-Security): Schutz eines Systems gegen Schäden durch zielgerichtete Angriffe, insbesondere in Bezug auf die Informationen, die es speichert, verarbeitet und kommuniziert.
- Sicherheitsziele:

1. Vertraulichkeit (Confidentiality)
2. Integrität (Integrity)
3. Verfügbarkeit (Availability)
4. Authentizität (Authenticity)
5. Verbindlichkeit (Non-repudiability)

Security Engineering

- Sicherheitsziele → Sicherheitspolitik → Sicherheitsarchitektur → Sicherheitsmechanismen
- Sicherheitspolitik: Regeln zum Erreichen eines Sicherheitsziels.
  - hierzu formale Sicherheitsmodelle:
  - IBAC, TE, MLS
  - DAC, MAC
- Sicherheitsmechanismen: Implementierung der Durchsetzung einer Sicherheitspolitik.
  - Zugriffssteuerungslisten(ACLs)
  - SELinux
- Sicherheitsarchitektur: Platzierung, Struktur und Interaktion von Sicherheitsmechanismen.
  - wesentlich: Referenzmonitorprinzipien
  - RM1: Unumgehbarkeit → vollständiges Finden aller Schnittstellen
  - RM2: Manipulationssicherheit → Sicherheit einerSicherheitspolitik selbst
  - RM3: Verifizierbarkeit → wohlstrukturierte und per Designkleine TCBs

Echtzeitfähigkeit

- Echtzeitfähigkeit: Fähigkeit eines Systems auf eine Eingabe innerhalb eines spezifizierten Zeitintervalls eine korrekte Reaktion hervorzubringen.
- Maximum dieses relativen Zeitintervalls: Frist d

1. echtzeitfähige Scheduling-Algorithmen für Prozessoren
   - zentral: garantierte Einhaltung von Fristen
   - wichtige Probleme: Prioritätsumkehr, Überlast, kausale Abhängigkeit
2. echtzeitfähige Interrupt-Behandlung
   - zweiteilig:asynchron registrieren, geplant bearbeiten
3. echtzeitfähige Speicherverwaltung
   - Primärspeicherverwaltung, VMM (Pinning)
   - Sekundärspeicherverwaltung, Festplattenscheduling

Adaptivität

- Adaptivität: Eigenschaft eines Systems, so gebaut zu sein, dass es ein gegebenes (breites) Spektrum nichtfunktionaler Eigenschaften unterstützt.
- Beobachtung: Adaptivität i.d.R. als komplementär und synergetisch zu anderen NFE:
  - Sparsamkeit
  - Robustheit
  - Sicherheit
  - Echzeitfähigkeit
  - Performanz
  - Wartbarkeit und Portierbarkeit

Adaptive Systemarchitekturen

- Zielstellungen:
  - Exokernel: { Adaptivität } ∪ { Performanz, Echtzeitfähigkeit, Wartbarkeit, Sparsamkeit }
  - Virtualisierung: { Adaptivität } ∪ { Wartbarkeit, Sicherheit, Robustheit }
  - Container: { Adaptivität } ∪ { Wartbarkeit, Portabilität, Sparsamkeit }

Performanz und Parallelität

- Performanz (wie hier besprochen): Eigenschaft eines Systems, die für korrekte Funktion (= Berechnung) benötigte Zeit zu minimieren.
- hier betrachtet: Kurze Antwort-und Reaktionszeiten

1. vor allen Dingen: Parallelisierung auf Betriebssystemebene zur weiteren Steigerung der Performanz/Ausnutzung von Multicore-Prozessoren(da Steigerung der Prozessortaktfrequenz kaum noch möglich)
2. weiterhin: Parallelisierung auf Anwendungsebene zur Verringerung der Antwortzeiten von Anwendungen und Grenzen der Parallelisierbarkeit(für Anwendungen auf einem Multicore-Betriebssystem).

Mechanismen, Architekturen, Grenzen der Parallelisierung

- Hardware:
  - Multicore-Prozessoren
  - Superskalarität
- Betriebssystem:
  - Multithreading(KLTs) und Scheduling
  - Synchronisation und Kommunikation
  - Lastangleichung
- Anwendung(sprogrammierer):
  - Parallelisierbarkeiteines Problems
  - optimaler Prozessoreneinsatz, Effizienz

Synergetische und konträre Eigenschaften

- Normalerweise:
  - Eine nichtfunktionale Eigenschaft bei IT-Systemen meist nicht ausreichend
  - Beispiel: Was nützt ein Echtzeit-Betriebssystem - z.B. innerhalb einer Flugzeugsteuerung - wenn es nicht auch verlässlich arbeitet?
- In diesem Zusammenhang interessant:
  - Welche nichtfunktionalen Eigenschaften mit Maßnahmen erreichbar, die in gleiche Richtung zielen, bei welchen wirken Maßnahmen eher gegenläufig?
  - Erstere sollen synergetische, die zweiten konträre (also in Widerspruch zueinander stehende) nichtfunktionale Eigenschaften genannt werden.
  - Zusammenhang nicht immer eindeutig und offensichtlich, wie z.B. bei: ,,Sicherheit kostet Zeit." (d.h. Performanz und Sicherheit sind nichtsynergetische Eigenschaften)

Notwendige NFE-Paarungen

- Motivation: Anwendungen (damit auch Betriebssysteme) für bestimmte Einsatzgebiete brauchen oft mehrere nichtfunktionale Eigenschaften gleichzeitig - unabhängig davon, ob sich diese synergetisch oder nichtsynergetisch zueinander verhalten.
- Beispiele:
  - Echtzeit und Verlässlichkeit: ,,SRÜ"-Systeme an potentiell gefährlichen Einsatzgebieten (Atomkraftwerk, Flugzeugsteuerung, Hinderniserkennung an Fahrzeugen, ...)
  - Echtzeit und Sparsamkeit: Teil der eingebetteten Systeme
  - Robustheit und Sparsamkeit: unter entsprechenden Umweltbedingungen eingesetzte autonome Systeme, z.B. smart-dust-Systeme

Überblick: NFE und Architekturkonzepte

- ✓... Zieleigenschaft
- (✓) ... synergetische Eigenschaft
- ✗... konträre Eigenschaft
- Leere Zellen: keine pauschale Aussage möglich.

Fazit: Breites und offenes Forschungsfeld → werden Sie aktiv!