Betriebssysteme

January 12, 2016

1 Aufgaben von Betriebssystemen

- Zwischen Hardware und Software
- Effiziente Nutzung der vorhandenen Ressourcen
- Stelle unendlich Virtuellen Speicher zur Verfügung
- Isolierte Addressräume für Anwendungen
- Verhindere Monopolisierung

2 Monolithe und Mikrokerne

Systemcalls sind im gefährlicher Bereich, eine Anwendung soll keinen direkten Zugriff auf die Hardware haben.
Anwendungen benutzen Libraries mit fremden Funktionen.
System Call Wrapper bietet eine Funktion, die die Systemaufrufe implementiert.
System calls liegen im Kernel, garantiert Schutz der Hardwarefunktionen, da nur über die System Calls Zugriff erlaubt ist.
Hardwareinterrupt: Gerät meldet per Leitung/Bit Interrupt. CPU rechnet fertig und bearbeitet diesen Interrupt.
System Calls sind Softwareinterrupts.
Bei der Ausführung von System calls wechselt das System in den privileged mode und auf den kernel stack. Im privileged mode ist Zugriff auf gefährliche Hardware erlaubt. (Supervisor/Usermode)
Nur das Betriebssystem darf im privileged mode laufen und tut dies auch immer, Anwendungen laufen im user mode.

2.1 Monolithen

Alle Funktionen des Betriebssystems liegen in einem großen Kernel. D.h. jeder System call wird in diesem einen Kernel ausgeführt.
Alle aktuellen Betriebssysteme sind Monolithen.
Verbesserung der Entwicklug durch Kernelmodule, die geladen/entladen werden können.

2.2 Mikrokerne

Für jede Funktionalität wird ein Mikroserver mit Mikrokern bereitsestellt (File,Grafik, Sound, Speicher etc.)
Anfragen werden u.U. durch alle Server geleitet, dabei entstehen für jeden Server Kontextwechse (Zeit!)
Kontextwechsel kosten viel Zeit, da bei jedem Kontextwechsel die Caches kalt werden, also keine Treffer besitzen.
Verbesserter Ansatz: Entwicklung als Mikrokerne, dadurch leichterer Entwicklung und Wartung. Nach erfolgreichem Testen werden diese Mikrokerne in einen Monolithen integriert und die Adressgrenzen aufgehoben.

3 Virtuelle Maschinen

Ermöglicht Migration von Virtuellen Maschinen wärend des Betriebs auf andere Hardware.
IBM entwickelte VM, da Maschinen zu teuer waren, um die neuen Versionen neben den Produktivsystem zu testen.
VMM (Hypervisor, Virtual Machine Monitor) läuft im priviligierten Modus und nimmt die Befehle des Gastsystems an und schickt das erwartete Ergebnis zurück. Das Gastsystem läuft im User-Mode und könnte diese Befehle nicht ausführen. Das Erkennen kritischer Instruktionen ist Aufwändig.
Hypervisor (Typ1 VMM): Direkt auf der Hardware, VMM ist quasi Betriebssystem
Typ2 VMM: Usermode Anwendungen (VirtualBox, etc.) Application Layer
Paravirutalisierung: Angepasste Gastsysteme, die kritische Stellen einem speziellen Hostsystem übergeben.
Ausblick: Anwendungen laufen jeweils in eigenen kleinen VM's
Immer mehr Anwendungen laufen in Managed Umgebungen (Runtime Environment etc.), also leicht virtualisiert.

4 Architekturansätze

4.1 Microsoft Singularity

Ansatz ein neues System zu entwickeln mit Microkernel, Zuverlässigkeit und modern.
Problem der aktuellen OS: Unmanaged Code und deren Zugriff auf den Adressraum. Wenn es nur Managed Software gibt, kann auch das OS managed sein. Speicherzugriff wird von Runtimes gemanaged. Dadurch müssen die Adressräume nicht isoliert werden, es gibt keine teuren Kontextwechsel.
Software isoliert Prozesse (SIP): Alles in einem Adressraum, aber Isoliert durch Software.
Interprozesskommunikation: Eigene Kommunikationssprache Sing# entwickelt. Zustandbasierter Ansatz. Kombiniert Vorteile von Mikrokernel aber umgeht die Schwächen.

5 Concurrency

Virtuelle Prozessoren: Erstelle für jeden virt. Prozessor einen eigenen Stack, sowie den CPU-Zustand (Register etc.)
Kontrollfluss siehe Syssoft: Running→Block: speicher Zustand in Speicher, erlaube einem ready-Prozess CPU-Zeit
Scheduler sendet regelmäßig interrupts, um monopolisierung zu verhindern.
Competition: Threads wollen CPU behalten.... Für immer..
Cooperation: Threads wissen, wann sie warten und geben CPU ab (per Semaphore/Mutex etc.)
Blockieren/Freigeben von Threads via Semaphore ist relativ teuer.
Prozessoren und Programmiersprachen bieten Interlocked-Methoden, die im Prozessor atomar ausgeführt werden.
Producer-Consumer/Reader-Writer Problem!
Spin-Locks: Erfordern atomaren Lese-/Schreibzyklus (Nicht Read/Write getrennt)
TAS(Test and Set) gibt den ursprünglichen Wert der Adresse zurück, bevor diese von TAS beschrieben wird.
Schnellste Art der Synchronisation, wenn auf many-core, durch paralleles busy-waiting.
Vorteil: Schnelle Behandlung der Scheduler-Queues
Nachteil: Kooperierende Threads, die parallel laufen sollten.
Aktuelle Betriebssysteme kennen keine Gang-Threads, Gruppen von Threads, die gleichzeitig laufen wollen.
Gedankenspiel: Es gibt keinen atomaren Lese-/Schreibzyklus
Grundlage: Es gibt einen kritischen Abschnitt
Implementierung enter() leave() des kritischen Abschnitts.
volatile: Variable kann sich durch anderen nicht jetzt compilierten code verändern, der Compiler soll es nicht wegoptimieren.
Deadlock: Es geht nicht weiter, aber alle Beteiligten tuen nichts
Lifelock: Es geht nicht weiter, aber alle Beteiligten brennen den Prozessor weg.

6 Echtzeitsysteme

Benötigt bestimmte Betriebssysteme (RTOS) und darin bestimmte Scheduler.
Voraussetzungen: Nach oben maximal abschätzbare Interupt-Zeit, minimale Kontextwechselzeit
Der Scheduler bekommt von einer Anwendung Zeitrahmen, indenen etwas erfüllt werden muss. Der Scheduler muss diese Zeitrahmen einhalten.
Echtzeitanwendungen haben: Ready time (ab wann darf bearbeitet werden), Deadline (bis wann MUSS es fertig sein), Δe (Ausführungszeit, worst case).
Echtzeitanwendungen dürfen nicht blockieren, dadurch kann man die Ausführzeit nicht abschätzen.
Preemption: Eine Ausführung kann unterbrochen werden, falls andere Prioritäten wichtiger sind. Achtung: Caches/Kontextwechsel
Static/Dynamic Scheduling: Statisch (Alle Faktoren genau bekannt)
Explicit/Implicit: Implicit (Prioritäten für Anwendungen), Explicit (Statischer Scheduling Plan, der fest steht)
EDF (Earliest Deadline First): Bearbeite Aufgabe mit nächstgelegener Deadline, Voraussetzung Cooperatives Anwendungsdesign
Nicht Preemptiv: Nicht optimal, preemptives EDF ist optimal.

Rate Monotonic Scheduling

Passt gut zu aktuellen Betriebssystemen, da diese statische Prioritäten verwenden.
Höhere Frequenz = Höhere Priorität. Je höher die Frequenz, desto kleiner ist die Ausführzeit (muss ja).
Klausur: Beispiel warum nicht Preemptive nicht geht!