Jan Schulz-Nigmann
03.04.2023

Zusammenfassung Betriebssysteme

Inhalt

Jan Schulz-Nigmann
03.04.2023

OS und Prozesse

Betriebssysteme

- Bietet eine Abstraktion der Hardware für Programme
- Verwaltet den Zugriff auf Ressourcen, versucht dieses möglichst Fair und Effizient
- Steuert die Ausführung von Programmen, sichert Schutz und Isolation zwischen Prozessen

Fundamentale Konzepte

- Abstraktionen: Prozesse, Threads, Sockets, Dateien, Disks, Geräte, …
- Mechanismen: open, create, fork, schedule, exec, close, send, …
- Policies: Wie Prozesse gescheduled werden, Speicher verwaltet wird etc

OS Protection Boundary

Prozesse können im user oder kernel mode ausgeführt werden, Priviledged operations sind nur im kernel mode möglich, dies wird durch die CPU Hardware sicher gestellt. Eine priviledged operation um User mode auszuführen führt zu einem Fehler.

Protection Rings:

- Ring 3: Programme
- Ring 2: Driver
- Ring 1: Driver
- Ring 0: Kernel

Heute nur noch Ring 0 und Ring 3.

System Calls

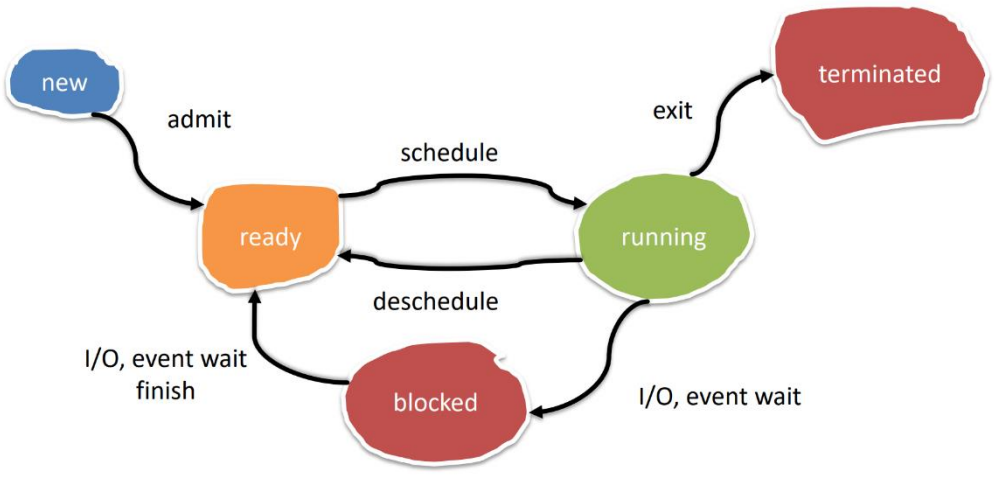
Programme benutzen System Calls um Services vom Kernel anzufragen: Hardware Services, Erstellen und Ausführen von neuen Prozessen, Kommunikation mit z.B. Prozess scheduling.

System calls bieten das Interface zwischen einem Prozess und dem Betriebssystem.

Was ist ein Prozess?

Ein Prozess ist die Abstraktion eines Programms während der Laufzeit. Prozess verwendet virtuellen Speicher: Struktur: text, data, heap (wächst von unten), stack (wächst von oben). Ein page table mappt die Adressen dann auf den physischen Speicher.

Ein Prozess hat folgende Zustände in denen er sich befinden kann:



Der Process Control Block (PCB) beinhaltet wichtige Informationen, die den Zustand des Prozesses beschreiben, wie: program counter, register inhalt, stack pointer, Speicherzuweisung, geöffnete Dateien, … Also alle Informationen die benötigt werden um den Prozess im gleichen Zustand weiterlaufen lassen zu können, falls er gestoppt wird

Multiprogramming

Mehrere Prozesse laufen in einem System parallel, im Konzept hat jeder Prozess seine eigene virtuelle CPU, in echt wechselt die CPU schnell von Prozess zu Prozess und erstellt die Illusion von Parallelismus (pseudoparallelism). Diesen Wechsel bezeichnet man als Kontextswitch.

Process Management (POSIX)

POSIX (Portable Operating System Interface) erstellt mit dem fork system call eine identische Kopie des Elternprozesses, dieser kann mit exec dann ein eigenes Programm laden. Der Elternprozess kann mit wait warten bis jedes seiner Kinderprozesse beendet ist. Prozesse können beendet werden in dem sie selbst exit aufrufen oder von ihrem Elternprozess mit dem kill-Befehl beendet werden.

Inter-Process Kommunikation (IPC)

Message passing-based IPC

Prozesse verwenden send und recieve um Nachrichten auszutauschen, diese werden in einer Message Queue verwaltet. Das Betriebssystem kontrolliert das Übertragen von Nachrichten, kommt aber mit einem zusätzlichen Overhead.

Shared memory-based IPC

Ein Bereich des physischen Speichers wird in beide virtuellen Speicher von zwei Prozessen gemappt. Das Betriebssystem muss also keine Kommunikation verwalten -> schneller aber komplizierter und Fehleranfälliger.

Scheduling und Threads

Scheduling

Multiprogramming bietet eine Möglichkeit um die CPU-Auslastung zu verbessern, wenn z.B. ein Prozess auf I/O Services wartet, kann ein anderer „nützliche“ Arbeit machen. Der Scheduler ist im niedrigsten Level des Betriebssystems implementiert, kümmert sich um Interrupts und scheduling, entscheidet also welcher Prozess als nächstes laufen darf.

Interrupts

Interrupts sind ein Signal an das Betriebssystem, welches diesem zeigt, dass ein Event aufgetreten ist, um das sich gekümmert werden muss, z.B.: input bereit, output fertig, error, clock interrupts

CPU checkt nach jeder Instruktion die interrupt-request Line, wenn ein interrupt vorhanden ist, wird der state des aktuellen Prozesses gespeichert, eine interrupt-handling Routine an einer speziellen Adresse ausgeführt, der Handler bedient den Interrupt und danach wird der state des vorherigen Prozesses wieder hergestellt. Dies muss effizient passieren, da single-user systeme hunderte interrupts und server sogar hunderttausende interrupts pro sekunde bearbeiten müssen.

Direct Memory Access (DMA)

Die CPU für Datenübertagung zu verwenden ist Verschwendung, daher werden DMA Controller verwendet, die Daten direkt aus dem Speicher in den Hauptspeicher verschieben können.

Kosten von Context Switching

Bei einem Kontextswitch wird der Kontext eines Prozesses (state) aus dem PCB im Speicher gesichert. Dann wird der Kontext des nächsten Prozesses aus dessen PCB geladen. Der typische Overhead für einen Kontextswitch sind ein paar Millisekunden.

Prozess Aktivität

Prozesse können entweder CPU-bound oder I/O-bound sein, ein CPU-bound Prozess wartet immer nur kurz auf den I/O Service und verbringt viel Zeit in einem CPU Burst. Ein I/O-bound Prozess wartet viel auf einen I/O Service und verbringt nur kurz Zeit in einem CPU Burst.

Wann Schedulen?

- Prozesserstellung: Eltern- oder Kindprozess?
- Prozessende: Welcher Prozess als nächstes? Wenn keiner, dann idle-Prozess
- Blockierter Prozess: Wenn ein Prozess z.B. auf I/O wartet, kann in der Zeit ein anderer Prozess laufen
- Interrupt: Ein interrupt kann dazu führen, dass nun ein anderer Prozess ausgeführt werden könnte, entscheiden, ob der aktuelle Prozess weiterlaufen soll, der unblocked Prozess oder ein anderer laufen soll.

Clock Interrupts

Reguläre Clock interrupts z.B. 250 Hz. Scheduling kann jeden Clock Interrupt oder jeden k-ten Clock interrupt passieren.

- Non-preemptive scheduling: lässt jeden Prozess bis zum block laufen oder bis er freiwillig die CPU freigibt
- Preemptive scheduling: Wenn ein maximaler Zeitslot erreicht ist, wird der Prozess gewechselt, benötigt eine clock

Scheduling Strategien

Verschiedene Umgebungen benötigen verschiedene Scheduling Algorithmen. Von diesen Umgebungen gibt es drei Stück: Batch, Interactive, Real-time

Generelle Ziele:

- Fairness: Jeder Prozess in derselben Prioritätsklasse bekommt die gleiche Prozesszeit
- Policy enforcement: Scheduling muss gewissen Regeln folgen, z.B. kritische Aufgaben müssen zuerst bedient werden
- Balance: Das ganze System soll beschäftigt sein, gesunder Mix zwischen CPU und I/O Bound Prozessen

Batch Scheduling

Geringe User-Interaktion, non-preemptive oder preemptive mit großen Zeitslots, weniger Kontextwechsel, bessere Performance.

Ziele:

- Throughput maximieren: Anzahl Prozess beendet pro Stunde
- Turnaround time minimieren: Durchschnittszeit, die ein Prozess benötigt
- Hohe CPU Auslastung: CPU arbeiten lassen anstelle von idle

Strategien:

- First Come, First Served (FCFS, FIFO): Einfach mit einer Linked List zu implementieren, potentiell Problematisch wenn mit verschiedenen Arten von Prozessen gemischt wird (Prozesse die schnell beendet werden müssen, können länger dauern, wenn erst langsame Prozesse an der Reihe sind)
- Shortest Job First (SJF): geschätzte Ausführungszeit muss vorher bekannt sein, bietet optimale durchschnittliche Turnaround Zeit
- Shortest Remaining Time Next (SRTN): preemptive Version von SJF, Prozess mit der kürzesten verbleibenden Zeit zuerst, lange Prozesse können lange warten

Scheduling in Interaktiven Systemen

Systeme mit on-line wartenden Nutzern, Netzwerk Server

Ziele:

- Response time minimieren: minimieren der Turnaround Zeiten der Prozesse, die der Nutzer aktiv benötigt
- Proportionalität sicherstellen: „Schnelle“ Jobs sollten schnell fertig sein, OK für große Jobs länger zu brauchen

Strategien:

- Round-Robin Scheduling: Jeder Prozess bekommt einen Zeitslot, genannt „Quantum“, einfach zu implementieren, Länge eines Quantums: Kontextswitch braucht viel Zeit, also sollte es wenige geben. Zu kurzer Quantum führt zu vielen „unnötigen“ Kontextswitches, zu langer Quantum führt zu schlechter Response Zeit
- Priority Scheduling: Prozesse sind in Queues aufgeteilt, Prozesse in einer Queue mit höherer Priorität werden zuerst bearbeitet
- Multiple Queues: Wie Priority, nur jede Queue bekommt Zeitslots, die niedrigeren Prioritätsqueues bekommen mehr Zeitslots. Jedes mal wenn ein Prozess durch den Zeitslot beendet wird, geht er in die nächst niedrigere Queue
- Shortest Prozess Next: Prozess mit der geschätzt kürzesten Ausführungszeit wird als nächstes ausgeführt. Sehr gut für Prozesse, die alternierenden Pattern folgen. Die geschätzte Ausführungszeit wird mithilfe vorheriger Beobachtungen angepasst.
  - $T_n = \alpha T_{n-1} + (1 - \alpha) T_{prev}$
- Guaranteed Scheduling: Jeder User von n Usern bekommt 1/n der CPU Zeit. Scheduler speichert das Verhältnis zwischen Zeitslot und tatsächlicher Laufzeit. Prozess mit dem niedrigsten Verhältnis läuft als nächstes
- Lottery Scheduling: Lottery Tickets werden zu jedem Prozess zugewiesen. Scheduler wählt zufällig ein Ticket aus. Die Wahrscheinlichkeit ist 1/n bei n Tickets. Kooperierende Prozesse teilen eventuell ein Ticket
- Fair-share Scheduling: Jeder User bekommt eine spezifizierte Zeit der CPU, unabhängig davon, wie viele Prozesse dieser User hat

Threads

Traditionell haben Prozesse nur ein Thread of execution. Es kann von Vorteil sein, mehrere Threads of control innerhalb eines Prozesses zu haben, diese werden vom Thread Scheduler gescheduled.

Threads können mehrere Aufgaben gleichzeitig ausführen, benötigt keine IPC, da die Threads den selben Speicherplatz und einige andere Parameter des Prozess states nutzen. Threads lassen sich deutlich schneller erstellen und zerstören als Prozesse. So kann man CPU-bound und I/O-bound Aufgaben überlappen.

Jan Schulz-Nigmann
03.04.2023

POSIX Threads

s. VL 2 Folie 45ff.

User-level vs Kernel-level Threads

User-level:

- Komplett im User space implementiert, der Kernel ist gar nicht involviert
- Kann in Betriebssystemen implementiert werden, die keine Threads unterstützen
- Prozesse brauchen private Thread Tabelle, diese beinhaltet die wichtigen Informationen des states des Threads
- Effektiver wenn keine häufigen blockierenden System calls auftreten
- Wie mit blockierenden System calls und page faults umgehen?

Kernel-level:

- Thread Tabelle vom Kernel verwaltet
- Thread erstellen und zerstören ist teurer, Threads können aber recycled werden
- Kann gut mit blockierenden System calls umgehen
- Was bei Prozess fork machen? Nur ein Thread oder alle duplizieren?

Hyprid:

- Auf einem Kernel Thread bauen mehrer User Threads auf

Prozessdatenstrukturen und Laufzeit Angriffe

Prozessdatenstrukturen

Assembly Sprache ist für jeden Typ von Prozessor unterschiedlich, x86-32bit Intel, x64-64bit Intel

Instruktionen: Kürzel operatoren: mov ecx 0x42

Endianess: Reihenfolge in der Bytes im Speicher gespeichert werden: Big Endian, most significat bytes first, little Endian, least significat bytes first

X86 und x64 nutzt little-endian, Internet-Protokolle verwenden Big-Endian.

CPU Register: Kleiner extrem schneller Speicher, der der CPU zur Verfügung steht, anderer Zugriff als Hauptspeicher. Von Registern gibt es verscheidene Arten: General-purpose, segment registers, program status und Kontrollregister, Instruction Pointer register, weitere Flags

General Purpose Register (x86):

- EAX: Accumulator Register: Akkumulator für Zwischenergebnisse und logische Ergebnisse
- EBX: Base Register: Basispointer für Speicherzugriffe
- ECX: Counter register: Zähler für Loops und String Operationen
- EDX: Data register (I/O pointer)
- ESI: Source Index pointer für String Operationen
- EDI: Destination Index für String Operationen
- EBP: Base Pointer (Pointer zu Daten im Stack)
- ESP: Stack Pointer

Speichersegmentierung

Programm wird ausgeführt: BS erstellt Address Space für Programm, in diesem Address Space ist das Programm, alle benötigten Daten und der Stack und Heap wird initialisiert.

Jan Schulz-Nigmann
03.04.2023

Segmente:

- Text: read-only, data und bss sind Schreibbar
- Data und bss Segmente für globale Variablen
- Data beinhaltet statisch initalisierte Daten und bss beinhaltet nicht-initialisierte Daten
- Text beinhaltet die Programm Instruktionen
- Stack ist eine Datenstruktur (LIFO) wächst nach unten
- Heap ist eine Datenstruktur (FIFO) wäscht nach oben

Stacksegment

Region auf dem Hauptspeicher der für Parameterübergabe reserviert ist, der Stack wird bei jedem Funktionsaufruf verwendet. Der Stack arbeitet mit einer LIFO Struktur und hat PUSH und POP Befehle. Der Stack ist in Frames unterteilt, jedes Frame sind die Daten die zu einem Funktionsaufruf gehören, ein Frame beinhaltet Argumente, lokale Variablen und die Return Address und den base Pointer.

Jeder Funktionsaufruf wird über die call Funktion gemacht, dabei wir die Return Address auf den Stack gepusht. Die Rückgabe passiert mit der ret Funktion, diese poppt die Return Address vom Stack und lädt diese in den Instruction Pointer (EIP).

Laufzeitattacken

Angriffe die während der Programmlaufzeit dieses verändern, dabei werden Software Exploits verwendet. Das ausführende Programm selbst wird dann zu Malware. Runtime Attacks können nicht trivial erkannt und verhindert werden. Alle Programm Binary checks sind ineffektiv, da die Programme erst während der Laufzeit angegriffen werden und Laufzeitchecks sind teuer.

Eine **Vulnerability** ist ein Fehler in einem System der ausgenutzt werden kann um vom eigentlichen Programmdesign abzuweichen: Denial of Service, Remote Code execution, Privilege escalation.

Arbitrary Code Execution: Erlaubt Angreifern seinen Code auszuführen um Zugriff auf das Opfersystem zu erhalten, z.B. root shell erstellen, neuen User hinzufügen, Netzwerk Port öffnen. Meistens wird der root shell angegriffen um dort Code auszuführenen.

Exploit: Eine Vulnerabilty verwenden um das Verhalten eines Systems zu ändern. 0-day Exploits sind Exploits, die noch nicht der Öffentlichkeit bekannt sind.

Buffer-Overflow Vulnerability: Ein Angreifer kann Arbitraty Code ausführen indem er die Return Address überschreibt. Über den Buffer hinaus in den Stack schreiben. Diese Vulnerabilities treten meistens wegen fehlenden Boundary Checks auf.

Buffer Overflow Protection: C bietet keinen Schutz, der User muss selbst darauf achten keine gefährlichen Funktionen zu verwenden, ansonsten muss der Programmierer manuell die Boundaries überprüfen. Es gibt aber auch generische Gegenmaßnahmenen: Address Space Layout Randomization (ASLR), Stack Canaries, Data Execution Prevention, Compiler protection: non-executable stack.

Code Injection: Ein neuen Knoten zu einem Kontrollflussgraphen hinzufügen, so kann Arbitrary Code ausgeführt werden.

Code Reuse: Ein neuen Pfad zu einem Kontrollflussgraphen hinzufügen, so können nur die vorhandenen Knoten verwendet werden, dafür braucht man statische Codeanalyse und muss den Code reverse engineeren.

Data Execution Prevention (DEP) verhindert Code Injection, daher warden heutzutage Angriffe meistens aus einer Kombination von Code Injection und Code Reuse aufgebaut.

Memory Management

No Memory Abstraction

- Alle Prozesse sehen und arbeiten auf den physischen Adressen. Verschiedene Prozesse können nicht gleichzeitig im Speicher sein. Z.B. Prompt-driven Systeme, Prozesse können „parallel“ laufen durch Swapping
- Absolute Adressen zu verwenden birgt einige Probleme
- Lösung: Statische Relocation: Alle Speicherreferenzen werden statisch überschrieben während das Programm lädt, dies verlangsamt den Ladeprozess und es werden zusätzliche Informationen benötigt, keine elegante/ flexible Lösung

Memory Management Voraussetzungen

- Protection: Ein Prozess sollte keinem anderen Prozess in die Quere kommen
- Fast translation: Speicherzugriffe sollten schnell gehen
- Fast context switch: Speicherupdate bei einem context switch sollte schnell sein

Memory Allocation

Fixed Partitioning

Physischer Speicher ist in festgelegte Partitionen unterteilt, eine Partition für jeden Prozess. Das Base Register muss dabei die logischen Adressen auf die physischen Adressen abbilden: Physische Adresse = Base Adresse + Logische Adresse. Das Base Register wird vom OS geladen.

Vorteile:

- Einfach zu implementieren
- Schneller Kontextwechsel

Probleme:

- Internal fragmentation: Nicht genutzter Speicher in einer Partition kann nicht von anderen Prozessen genutzt werden
- Wie die richtige Partitionsgröße wählen?

Variable Partitions

Physischer Speicher ist in variabel große Partitionen unterteilt, braucht ein Base Register und ein Limit Register. Bietet Prozessisolation mithilfe des limit registers.

Probleme:

- External fragmentation des physischen Speichers: Job laden und entladen erzeigt „Löcher“ im Speicher

Virtual Memory

Prozesse greifen auf den Speicher mittels einer virtuellen Adresse zu. Virtuelle Adressen ermöglichen, dass die physischen Adressen bei jedem Prozessstart eine andere sein kann. Programme können ausgeführt werden, ohne dass der gesamte virtuelle Speicher physisch vorhanden ist, so kann ein Programm mehr Speicher adressieren als eigentlich vorhanden, implementiert durch Paging oder Segmentation.

Segmentation

Naturliche Erweiterung von Variable Size Partitioning, Variable Size: 1 Partition pro Prozesse, Segmentation: mehrere Segmente per Prozess. Segmenttabelle wird genutzt um die Segmente auf die physischen Adressen abzubilden, jedes Segment hat dabei eine Base Adresse und ein Limit. Segmenttabelle Basis Register (STBR) zeigt dabei auf die Adresse der Segmenttabelle im Speicher und das Segmenttabelle Länge Register (STLR) gibt die Anzahl der Segmente in einem Prozess an.

Vorteile:

- Keine interne Fragmentierung
- Mehrere unabhängige, geschützte Address Spaces in einem Prozess
- Die einzelnen Segmente müssen im physischen Speicher nicht aufeinander folgen, der Inhalt der Segmente sollte allerdings aufeinander folgen

Probleme:

- Externe Fragmentierung: Beim laden und entladen von Prozessen entstehen viele kleine Stücke

Memory Paging

Speicher in gleichgroße Blöcke teilen, jeder Block wird einzeln gemanaged. Jede Page im virtuellen Speicher kann auf jede Page im physischen Speicher gemappt werden. Jede Zuweisung hat bestimmte Rechte (R;W;X). Virtuelle Pages müssen nicht auf physische Pages abgebildet werden.

Page Sharing: Mehrere Prozesse können den selben Code teilen und ausführen, dieser geteilte Speicher kann auch zur IPC verwendet werden. Eine Shared Page kann in verschieden virtuellen Adressen sein aber auf der selben physischen Adresse.

Die Logische Adresse hat zwei Teile: Page number (PN) und offset, die Pagetabelle übersetzt die Page number in die Frame number (FN).

Pagetabellen können als Set von dedizierten Registern (nur für kleine Tabellen) oder im Hauptspeicher umgesetzt werden (längere Zugriffszeit, TLB nutzen).

Paging mit Translation Look-aside Buffer (TLB): Memory Management Unit (MMU) speichert einen Cache an zuletzt genutzten Zuweisungen der Pagetabelle

Hierarchical Paging

Pagetabellen können sehr groß werden, es ist schwer einen kontinuierlichen, so großen Speicher zu finden, daher verwendet man Hierarchical Paging.

2-level Paging:

- Pagetabelle wird in Sub-Pagetabellen aufgeteilt
- Logische Adresse wird in 3 Teile geteilt: PN in äußerer Pagetabelle, PN in Pagetabellen Page und Page Offset
- Größere Speichereffizienz

Real World Address Paging (4-level Paging)

- Wie 2-level Paging nur mit 4 leveln an Pagetabellen
- Noch großere Speichereffizienz aber längere Latenz
- Auflösen von virtuellen Adressen: Page Walk (Suchen in den unterschiedlichen Ebenen)

Page Protection

- Pagetabellen beinhalten auch acces control Informationen
- Implementiert mit Protection bits zu jedem Frame
- Auf Invalid Pages zugreifen führt zu einem Page fault
- R/W oder Read-only bits, auf eine Read-only Page zu schreiben führt zu einem Page fault

Geräte I/O

Ebenen: Hardware -> Interrupt Handlers -> Device Driver -> Device-independent operating system software -> User-level I/O software

Gerätetypen:

- Block Devices: Geräte die einen Block an Daten auf einmal schreiben /lesen
- Character Devices: Geräte die mit einem Characterstream arbeiten, nicht adressierbar und keine seek Operation

Device Controller: Elektronischer Teil eines I/O Geräts, konvertiert seriellen bit stream in Byte-Blöcke, führt Error-Correction durch, ist für low-level Kontrolle des Geräts zuständig und wird mittels Control Registers gesteuert.

Interrupts

Geräte erzeugen einen Interrupt indem ein Signal auf der zugewiesenen Bus-Linie gesetzt wird. Der Interrupt Controller entscheidet welcher Interrupt als erstes bearbeitet wird. Der Controller platziert dann eine Nummer an den Address-Bus, welches Gerät Aufmerksamkeit benötigt und erzeugt den Interrupt. Die gesendete Nummer fungiert als Index in einer Tabelle, der Interrupt Vektor beinhaltet die Program counter Werte, für die Routinen die mit dem Interrupt umgehen sollen.

Interrupt Handling

Wenn eine Interrupt Routine startet, speichert die CPU den State um den aktuellen Prozess danach weiterlaufen zu lassen und bestätigt den Interrupt. Der aktuelle Prozess-State kann in den internen Registern, dem Prozessstack oder dem Kernelstack gespeichert werden.

Moderne CPUS sind stark pipelinded und meist superskalar (mehrere Instruktionen parallel). Der aktuelle Wert im Program Counter kann also die Grenze zwischen ausführenden und nicht-ausführenden Instruktionen nicht immer darstellen

Precise Interrupts:

- Ein präziser Interrupt lässt die Maschine in einem gut definierten Status.
- Der Programm Counter wird gespeichert
- Alle Instruktionen vor dem PC sind fertig
- Alle Instruktionen nach dem PC sind nicht ausgeführt
- Der Ausführungsstatus der Instruktion am PC ist bekannt

Imprecise Interrupt:

- Benötigt mehr Informationen über den internen Status der CPU um den das unterbrochene Programm weiter auszuführen: Komplizierter und langsamer
- In manchen Systeme gibt es Interrupts die Precise sein müssen und andere die Imprecise sein können
- Intel x86 bietet Precise Interrupts um Backward Compatability zu gewährleisten, dies braucht sehr komplizierte Interrupt logic in der CPU Hardware

Jan Schulz-Nigmann
03.04.2023

I/O Software

Aspekte von I/O Softwaredesign:

- Device independence: I/O Zugriff ist unabhängig vom Gerätetyp
- Uniform naming: Erlaubt Zugriff zu Geräten mittels Identifiern, die unabhängig vom Gerät sind
- Error handling: Erlaubt transparente Recovery von Fehler bei möglichst niedrigem Level
- Synchronous vs Asynchronous: Bietet synchrone I/O Abstraktion zu Programmierern
- Buffering: Geräte-Datenrate von System-Datenrate entkoppeln

Programmed I/O

- I/O programmiert durch Software (z.B. in C), Drucker
- Die CPU wartet darauf, dass das Gerät bereit wird und führt dann die nächste I/O Operation durch

Interrupt-Driven I/O

- Der Prozess, der eine I/O Operation angefordert hat, wird blockiert bis die I/O Operation angeschlossen ist, in der Zwischenzeit können andere Prozesse laufen
- Wenn das Gerät fertig ist, generiert es einen Interrupt, der die Blockade des wartenden Prozesses aufhebt

I/O using DMA

- DMA controller ist programmiert um Daten in ein Gerät zu füttern, ohne die CPU zu verwenden.
- Interrupt wird durch den DMA nur generiert, wenn die gesamte I/O Operation beendet ist

Device Drivers

Jeder Gerätecontroller stellt Geräteregister zur Verfügung über die Kommandos an das Gerät gegeben werden können und der Gerätestatus gelesen werden kann.

Der gerätespezifische Code, der diese Register zum Steuern des Gerätes benutzt heißt Device Driver. Dieser ist typischerweise vom Gerätehersteller geschrieben und wird mit dem Gerät zur Verfügung gestellt. Heutzutage dynamisch in den Kernel geladen als Teil des OS Kernels.

Rolle des Gerätetreibers

- Akzeptiert abstrakte Lese und Schreib-Anfragen von höher liegenden Softwareebenen und führt die entsprechenden Aktionen aus um die Anfrage zu beantworten
- Input Parameter überprüfen
- Abstrakte Anfragen auf konkrete übersetzten
- Wenn das Gerät verwendet wird, Anfragen zurückstellen und später beantworten
- Gerätestatus initialisieren
- Gerät kontrollieren durch Kommandos in control Register schreiben und Gerätestatus lesen
- Wenn der Treiber auf I/O Operationen warten muss, blockiert dieser bis zum passenden Interrupt
- Fehler korrigieren und Output an die höhere Softwareebene weitergeben
- Gerätetreiber müssen wiedereintretend sein: Ein Gerätetreiber kann durch einen I/O Interrupt blockiert werden und danach wieder durch den Interrupt Handler aufgerufen werden

- Error handling: Geräte können während einer Operation unavailable werden, dann muss der Treiber diese Prozesse abbrechen können

Device-Independent I/O Software

- Teile der I/O Software sind unabhängig von den Geräten
- Typische Aufgaben werden durch diese Software behandelt: Uniform interfacing, Buffering, Error reporting, Allocating und Freigeben von bestimmten Geräten, Zurverfügungstellen von Geräteunanbhängiger Blockgröße

Uniform Interfacing für Gerätetreiber

- Standartisierung des Interface für I/O Geräte erlaubt einfaches hinzufügen von neuen Geräten an ein System ohne das OS modifizieren zu müssen
- Für jede Klasse von Geräten ist ein Satz an Funktionen gegeben, an die sich die Geräte halten müssen.
- Uniform Naming von Geräten: Geräteunabhängige Software weißt Gerätename zu Hardware-Gerätename zu

I/O Buffering

Arten:

- Unbuffered I/O führt zu vielen Interrupts, jeder Character wird mit einem Interrupt bearbeitet
- User-level buffer reduziert die Nummer der generierten Interrupts, was passiert beim Entfernen des Buffers aus dem Speicher durch Paging?
- Kernel-level buffer hat extra Speicher um I/O zu buffern. Umgeht das Paging Problem, was passiert wenn der Buffer voll ist?
- Double buffering in kernel kopiert den vollen Kernel buffer in den User buffer

User-Space I/O Software

User-space Bibliotheken bieten notwendige Routinen für I/O Anfragen.

Spooling System:

- Lösungen für geteilte Zugriffe auf I/O Devices
- Spezieller Prozess „Deamon“ hat exklusiven Zugriff auf I/O Gerät
- Regulärer Prozess generiert Dateien und platziert diese im Spooling Verzeichnis
- Deamon Prozess bearbeitet Dateien in Spooling Verzeichnis und sendet diese an das Gerät

Swapping

Wenn aktive Prozesse mehr Speicher benötigen als zur Verfügung steht, müssen manche Prozesse auf die Festplatte geswappt werden. Speicher eines Prozess der gerade nicht läuft wird geswappt, damit der Speicher anderen Prozessen zur Verfügung steht. Dafür wird meistens ein spezieller Speicherplatz verwendet: swap file oder swap partition.

Zwei Methoden zum Verwalten von freiem Speicher:

- Memory Bitmaps: Speicherbesetzung wird in einer Bitmap gespeichert, die Größe der Blöcke beeinflusst die Größe der Bitmap, große Blockgröße verschwendet womöglich Speicher und das finden von Blöcken mit einer gewissen Länge kann langsam sein

- Linked Lists: Segmente von benutzten und unbenutzten Speicher werden in einer Linked List gespeichert, auswechseln von Prozessen ist einfach

Allokationsstrategien:

- First fit: Erstes Segment, welches groß genug ist, simpel und einfach zu implementieren
- Next fit: Wie First fit nur sucht weiter an der Stelle an der er aufgehört hat
- Best fit: Gesamte Liste durchsuchen nach dem kleinsten Segment in das der Prozess passt, langsamer, verschwendet Speicher
- Worst fit: Füllt größtes Segment mit dem Prozess um kleine Löcher zu vermeiden
- Quick fit: Extra Listen mit Löchern, macht finden von Löchern in der benötigten Größe schnell, Segmente kombinieren nach auswechseln ist aufwendig

Paging

Beim swappen wir der gesamte Speicher eines Prozesses ausgewechselt. Bei virtuellen Speichersystemen, die Paging verwenden, werden nur Teile des Prozesses ausgewechselt, so müssen nur die Pages im Speicher behalten werden, die der Prozess gerade benötigt. Andere Pages können ausgetauscht werden.

Page Fault

Ein Page Fault tritt auf, wenn ein Prozess auf eine Page zugreift, die nicht zugewiesen ist.

Folgende Fehlertypen existieren:

- Minor Page fault: Page ist im physischen Speicher vorhanden aber nicht im Address Space des Prozesses vorhanden. Kann behoben werden, in dem der Page Table geupdated wird.
- Major Page fault: Page ist nicht im physischen Speicher, muss also von der Festplatte geladen werden
- Segmentation fault: Prozess versucht auf eine invalid Adresse zuzugreifen, führt meistens dazu, dass der Prozess vom OS beendet wird

Demand Paging: Speichermanagmentansatz keine Pages vorgeladen werden, es wird also für jede Page ein Major Page fault geworfen und die Page wird dann geladen, wird auch lazy loading genannt. Verhindert unnötiges laden von Pages in den Hauptspeicher.

Page Replacement

Anzahl der physischen Page Frames ist limitiert. Wenn alle Page Frames belegt sind, muss das OS bei einem Page Fault entscheiden, welche Page ausgetauscht werden soll. Wenn die auszutauschende Page verändert wurde, muss diese erst auf der Festplatte gespeichert werden, sonst kann die Page einfach überschrieben werden.

Page Table Entries

Present bit: Ob die Page im physischen Speicher vorhanden ist oder nicht

Protection bits: Zugriffsrechte auf die Page (R;W;X)

Modified bit: Ob die Page bearbeitet wurde oder nicht

Referenced bit: Speichert, ob eine Page für lesen oder schreiben verwendet wurde

Caching disabled bit: Legt fest, das seine Page nicht gecached werden soll

Page Reference Bookkeeping: Page replacement Algorithmen verwenden Buchhaltungsinformationen über die Page Frames, am Anfang sind R und M bits 0, R wird bei jedem clock tick wieder auf 0 gesetzt.

Page Replacement Algorithmen

Optimal Page Replacement: Die Page austauschen, die in Zukunft am längsten nicht benötigt wird, dieser Algorithmus ist aber unmöglich, da man nicht weiß, wann eine Page benötigt wird. Gut zum Evaluieren von anderen Algorithmen.

Not Recently Used (NRU): Bei einem Clock Interrupt werden die Pages in eine von 4 Klassen klassifiziert: Class 0: R=0, M=0, Class 1: R=0, M=1, Class 2: R=1, M=0, Class 3: R=1, M=1. Bei einem Page Fault wird zufällig eine der Pages in der niedrigsten Klasse ausgetauscht.

First-In, First-Out (FIFO): Bei einem Page Fault wird die Page, welche am längsten im Speicher ist getauscht, kann einfach mit einer Linked List implementiert werden, selten genutzt, da alte Pages trotzdem häufig genutzt werden könnten.

Second-Chance: Variation von FIFO, wenn R=1 wird die älteste Page nicht entfernt, sondern neu in die Linked List angehängt und R wird wieder auf 0 gesetzt, wenn alle R auf 1 gesetzt sind, funktioniert Second Chance exakt wie FIFO.

Clock: Page Frames sind in einer Kreis-Liste angeordnet mit einem Zeiger. Bei einem Page Fault wird die aktuelle Page angeschaut und wenn R=0, dann ausgetauscht, ansonsten wird R auf 0 gesetzt und die nächste Page geprüft.

Least Recently Used (LRU): Bei einem Page Fault wird die Page getauscht, welche am längsten nicht verwendet wurde. Realisiert mit einer Linked List, wobei die zuletzt genutzte Page am Anfang und die als letztes benutzt am Ende ist. Die List muss aber bei jeder Speicherreferenz geupdated werden. Zeitaufwendig und braucht spezielle Hardware.

Not Frequently Used (NFU): Jedes Page Frame ist mit einem Zähler ausgestattet, bei jedem Clock Interrupt wird der Counter mit R erhöht (+0 oder +1). Wenn ein Page Fault auftritt, wird die Page mit dem geringsten Zählerstand ausgetauscht. Problem: Pages die früher viel genutzt wurden und jetzt gar nicht mehr, können sehr lange im Speicher verweilen.

Ageing: Zähler wie bei NFU aber bei jedem Clock Interrupt wird der Zähler um 1 nach rechts geshifted und danach R auf der linken Seite angehängt. Bei einem Page Fault wird das Page Frame mit dem niedrigsten Zählerwert getauscht. Die Bitbreite des Zählers limitiert die Anzahl der Clockticks, die gespeichert werden.

Working Set: Die meisten Prozesse verwenden locality of reference, also das jeder Prozess zu jeder Zeit nur ein kleines Subset an Speicherpages verwendet. Die aktuell verwendeten Pages werden das Working Set genannt. Wenn das Working Set im Speicher vorhanden ist, läuft der Prozess mit wenigen Page Faults. Ein Paging System versucht also zu tracken, welche Pages im Working Set sind um diese vor Beginn eines Prozesses zu laden. Das nennt man Prepaging.

Working Set festlegen:

- 1. Approximation durch ansehen der zuletzt verwendeten Pages des Prozesses, braucht also in der Pagetabelle einen weiteren Eintrag.
- 2. Wenn ein Page Fault auftritt, werden die Pagetabellen Einträge gescannt.
    - 2.1 Wenn R=1: Die letzte Zugriffszeit auf aktuelle updaten und R=0 setzten.

  - 2.2 Wenn R=0: Wenn die Page „alt“ ist (age > τ) also nicht im Working set ist, entfernen, wenn keine der Pages „alt“ ist, die mit der letzten Zugriffszeit entfernen.
  - 2.3 Wenn alle Pages genutzt werden, also R=1, random Page entfernen.

Thrashing: Thrashing kann auftreten, wenn das Working Set größer als der vorhandene Speicher ist. Dies führt dazu, dass Pages die bald verwendet werden nicht geladen werden und dadurch viele Page Faults entstehen. Ein thrashing Prozess wird also die ganze Zeit auf Pages warten und so kaum was arbeiten.

WSClock: Kombination aus Working Set und Clock, Pagetabellen Einträge sind in einer Clock angeordnet, jeder Eintrag beinhaltet letzte Zugriffszeit, R und M bits. Wenn ein Page Fault auftritt, wird der Eintrag auf den der Zeiger zeigt betrachtet:

- Wenn R=1 wird der Zeiger auf den nächsten Eintrag gesetzt und R auf 0 gesetzt.
- Wenn R=0, wird das alter des Eintrags untersucht:
  - 1. Wenn Page alt (age > τ) und die Page „dirty“ ist (M=1), wird ein Page write ausgeführt und der Zeiger geht einen Eintrag weiter.
  - 2. Wenn Page alt ist (age > τ) und die Page „clean“ ist (M=0) wird sie ausgetauscht.
- Was passiert, wenn der Zeiger einmal im Kreis gelaufen ist und keine Page getauscht wurde?
  - 1. Wenn mindestens eine Page einen Page write angestoßen hat, läuft der Zeiger so lange weiter bis er eine cleane Page mit R=0 findet, irgendwann wird der Page write fertig sein und dann kann eine Page ausgetauscht werden.
  - 2. Wenn keine Writes angestoßen wurden (also keine Page alt ist), wird eine cleane Page ausgetauscht, gibt es davon auch keine, wird die aktuelle Page ausgewechselt

Memory Allocation Policies

Memory Allocation policy legt fest, wie der vorhandene physische Speicher unter der Prozessen aufgeteilt wird.

Local Page Replacement: Die zu entfernende Page wird aus den Pages ausgewählt, die zu dem Prozess gehören.

Global Page Replacement: Die zu entfernende Page wird aus allen Pages aller Prozesse ausgewählt. Die Anzahl der Frames die also zu einem Prozess gehören ist dynamisch.

Global allocation funktioniert besser wenn die Working Set-Größe stark variiert. Local allocation endet in Thrashing, wenn die Working Set-Größe wächst und verschwendet Speicher, wenn die Working Set-Größe schrumpft.

Allocation Größe festlegen:

- Periodic fixed allocation: Jeder active Prozess erhält einen Bruchteil der verfügbaren Page Frames, beachtet nicht, dass Prozesse verschieden Größen haben können
- Proportional allocation: Passt die Größe Proportional an die Prozessgröße an
- Anpassen durch Page Fault Frequency (PFF): Globaler Algorithmus kann die pro-Prozess PFF überwachen und die Allokationsgröße erhöhen, wenn zu viele Page Faults auftreten und verkleinern, wenn keine/wenige Page Faults auftreten

Page Fault Frequency (PFF)

- Basiert auf der Anzahl der Page Frames die zu einem Prozess gehören
- PFF > A: inakzeptabel große PFF -> allokation vergrößern
- A > PFF > B: gutes PFF level -> Allokationsgröße beibehalten

- PFF < B: Prozess hat zu viel Speicher, Allokation verkleinern

Load Control: Wenn die kombinierte Größe der Working Sets der Prozesse größer als der verfügbare Speicher ist, fängt das System an zu trashen. Um das zu beheben, muss Speicher freigegeben werden, also gesamten Prozess austauschen und dessen Page Frames freigeben bis das System nicht mehr trasht.

Caches

Caches sind Speicher die zwischen der CPU und den Hauptspeicher sind. Es gibt L1 (am kleinsten un schnellsten), L2 (mittel groß und mittel schnell) und L3 (am größten und am langsamsten) um die Speicherzugriffszeiten zu verbessern. Jeder Core hat einzelne L1 und L2 Caches und teilen sich den L3 Cache.

Split Caches: Bei Split Caches werden zwei getrennte Speicher für Instruktionen und Daten verwendet.

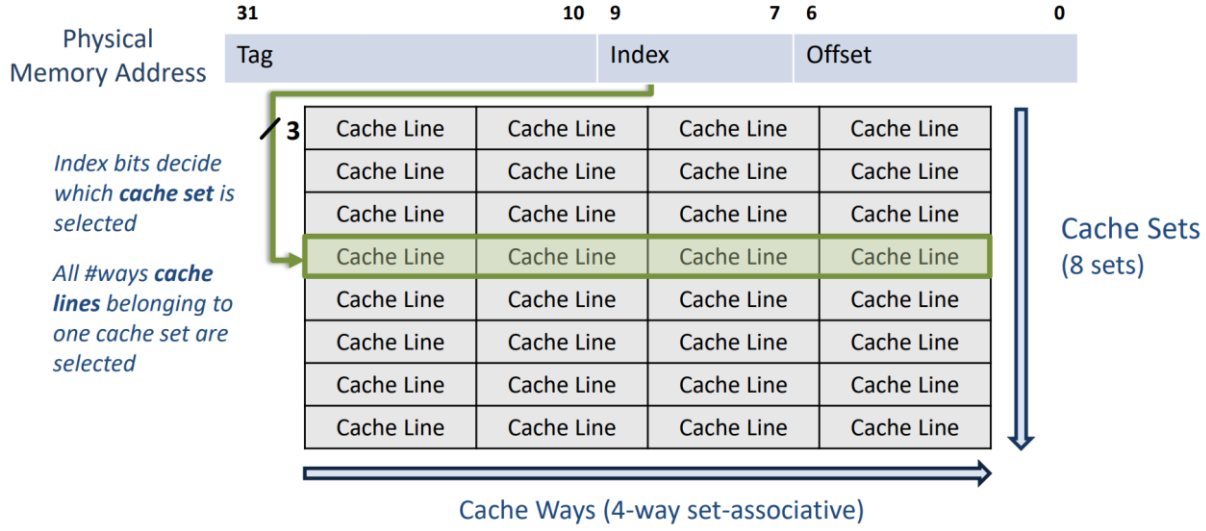
Unified Caches: Bei Unified Caches werden Instruktionen und Daten im selben Speicher abgelegt.

Inclusive Caches: Daten werden im Core exklusiven Cache und in den shared Caches gespeichert.

Exclusive Caches: Daten werden entweder im Core exklusiven Cache oder in den Shared Caches gespeichert.

Non-exclusive Caches: Bei nicht-exklusiven (oder Teil-Inklusiven) Caches können Daten in dem Core exklusiven Cache und den Shared Caches gespeichert werden, müssen es aber nicht.

Cache Architektur



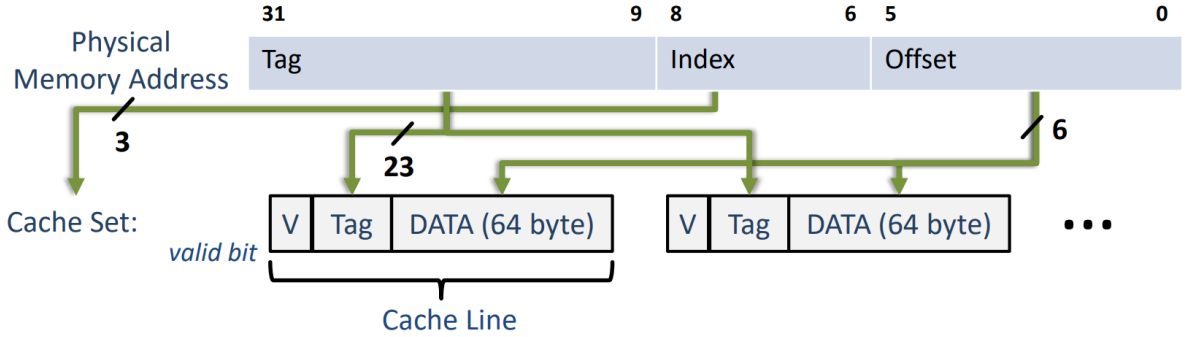
Cache Assoziativität

Set-associative cache: Bei einem x-way set-assoziativen Cache können Daten einer bestimmten Speicheradresse in x verschiedenen Cache Lines (oder Ways) gespeichert werden, diese formen ein Cache Set. S.o.

Fully associative cache: In einem voll-assoziativen Cache können Daten einer bestimmten Speicheradresse in allen der y cache Lines des Caches gespeichert werden. Man kann sich das vorstellen wie ein Cache Set mit y Ways. Voll-assoziative Caches werden nur bei kleinen Caches nah an der CPU verwendet, sie verursachen wenige Cache misses aber verbrauchen mehr Strom.

Directly-mapped cache: Bei einem direkt-mapped Cache können Daten einer bestimmten Speicheradresse nur in einer einzigen Cache Line gespeichert werden. Man kann sich das vorstellen wie ein Cache der nur einen Way pro Cache set hat.

Cache Lines



Der Cache Controller vergleicht die Tag bits von allen #ways Cache Lines (mit valid bit=1) mit den Speicheradress-Tag bits.

Match: Der Cache beinhaltet bereits die Daten der Speicheradresse (cache hit)

No Match: Der Cache beinhaltet die Daten der Speicheradresse nicht oder ist nicht mehr valid (cache miss)

Bei einem Cache hit werden die Offset Bits verwendet um das korrekte Byte im DATA Block zu finden.

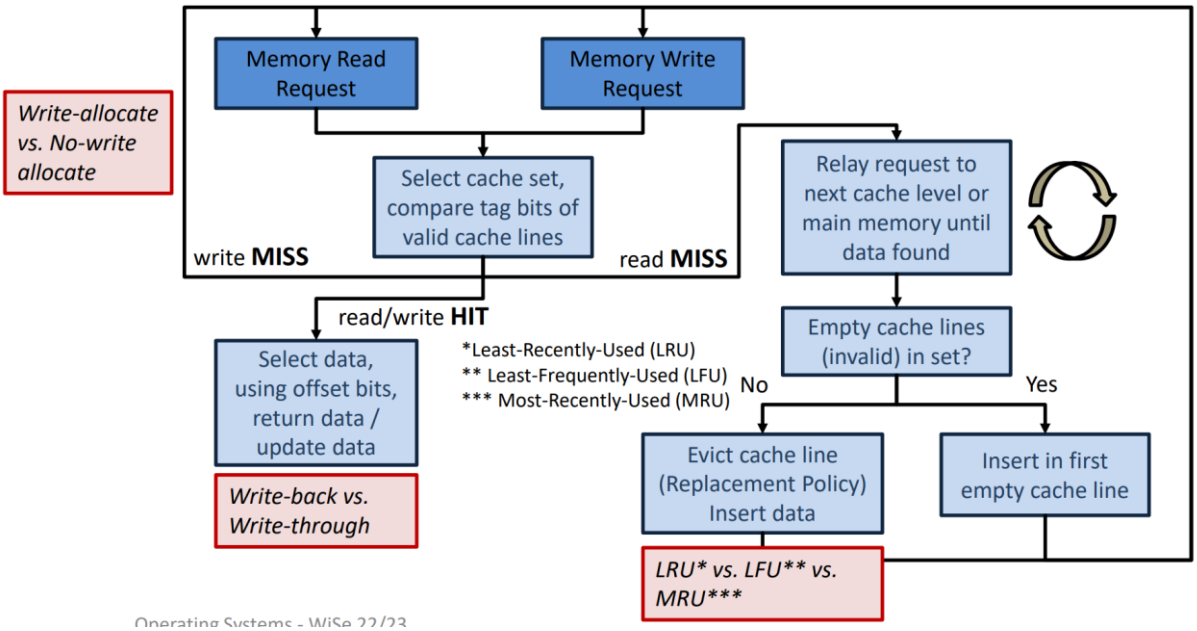
Cache Adressing

Physically Indexed Physically Tagged (PIPT): Bei PIPT wird der Index und die Tag Bits aus der physischen Speicheradresse entnommen. Der Prozessor muss zuerst die Adresse von der virtuellen in die physische übersetzten: mehr Latency. PIPT ist deswegen nicht bei Caches nah an der CPU verwendet.

Virtually Indexed Virtually Tagged (VIVT): Bei VIVT wird der Index und die Tag Bits aus der virtuellen Adresse entommen, es ist also keine Adressenübersetzung nötig. Da virtuelle Adressen aber nur in einem Prozess eindeutig sind, kann es sein, dass die selbe virtuelle Adresse zu verschiedenen physischen Adressen zeigt (Homonym Problem). Auch verschiedene virtuelle Adressen können auf eine physische Adresse zeigen (Synonym Problem). Das OS oder der Cache Controller muss sich um diese Probleme kümmern.

Virtually Indexed Physically Tagged (VIPT): Bei VIPT wird der Index aus der virtuellen Adresse entnommen und die Tag Bits aus der Physischen. Der Prozessor kann die Adressübersetzung dann parallel mit dem Cache Controller machen, da dieser bereits indizieren kann, während die CPU übersetzt. Der physische Tag verhindert Homonyms, Synonyms sind aber noch möglich.

Cache Control Flow



Write-hit Policies

Write-back: Bei einer Writeback Policy werden die Daten nur in der Cache Line geändert und nicht im Hauptspeicher. Der Hauptspeicher wird somit inkonsistent, daher wird zu einer Cacheline ein Dirty Bit hinzugefügt um anzuzeigen, dass eine Cacheline bearbeitet wurde aber noch nicht im Hauptspeicher vorhanden ist. Diese Dirty Bits sind vorallem wichtig, wenn es darum geht Cachelines aus dem Cache zu entfernen, sind diese dirty, müssen sie erst in den Hauptspeicher geladen werden.

Write-through: Die Daten werden synchron in der Cacheline und dem Hauptspeicher geändert, es ist kein Dirty Bit notwendig.

Write-miss Policies

Write-allocate: Bei der Write-allocate Policy (meistens in Kombination mit Write-back) wird bei einem Cache Miss eine Exception ausgelöst, die dafür sorgt, dass die Daten aus dem Hauptspeicher in den Cache geladen werden.

No-write allocate: Bei dieser Policy (meistens in Kombination mit Write-through) führt ein Cache-Miss zu einer Änderung auf dem Hauptspeicher, es werden keine Daten in den Cache gespeichert.

Replacement Policies

Least Recently Used (LRU): Der Cache Controller entfernt immer die älteste Cacheline, hierzu muss aber das Alter der Lines extra gespeichert werden.

Least Frequently Used (LFU): Der Cache Controller zählt, wie oft eine Cacheline verwendet wurde und entfernt die, welche am wenigsten benutzt wurde.

Most Recently Used (MRU): Der Cache Controller entfernt immer die Cacheline mit dem jüngsten Alter.

Memory Mappings

Memory Mapped Files

TODO Copy-on-Write

I/O mit I/O Ports

Zugriff auf Gerätekontrollregister mittels I/O Ports, jedes Register hat eine Port-Nummer. Um auf die I/O Ports zuzugreifen, gibt es spezielle I/O Instruktionen.

Memory-Mapped I/O

Jedem Kontrollregister ist eine einzigartige Speicheradresse zugewiesen, kein echter Speicher wird diesen Adressen zugewiesen. Erlaubt schnellen Zugriff auf Inputs und Outputs des Geräts, da die Daten nicht aus den Geräteregistern kopiert werden müssen.

Vorteile:

- Benötigt keine extra Instruktionen
- Schutz eines I/O-Geräts kann durch Memory Mapping erreicht werden
- Testen der Geräteregister geht schneller, da diese direkt auf der Speicheradresse durchgeführt werden können

Nachteile:

- Caching ist problematisch, da es für Pages die auf ein Kontrollregister gemappt sind ausgeschaltet werden muss und die Hardware verkompliziert
- Speicheradressierung ist bei Geräten mit mehreren Buses komplizierter

Seperate Memory Spaces

Wenn der Address Space klein ist, kann es schwierig sein die Prozesse und deren Daten in den Speicher zu bekommen. Früher haben Systeme getrennte Speicher für Instruktionen und Daten verwendet. Die beiden Address Spaces können dann getrennt gepaged werden. L1-Cache verwendet immernoch getrennte Address Spaces.

Direct Memory Access (DMA)

Anstatt I/O Operationen über die CPU zu machen die Daten direkt an das Gerät senden. Die CPU beauftragt den DMA Controller mit dem Datentransfer und kann dann mit anderen Aufgaben weitermachen, dazu gibt die CPU die Start und Ziel-Adresse, die Länge und Kontrollkommandos an den DMA Controller. Der DMA Controller teilt dann den Geräten mit die Daten zu übertragen bis die Daten fertig übertragen sind. Der CPU wird per Interrupt Bescheid gegeben, wenn der Transfer fertig ist. Benötigt internen Geräte Buffer um Checksums zu überprüfen und man muss nicht auf den Memory Bus warten, da die Daten stetig kommen. DMA ist aber nicht immer nützlich, wenn z.B. die CPU deutlich schneller ist und Komplexitätsreduzierung in eingebetteten Geräten.

Operating Modes:

- Word-at-a-time mode: DMA verwendet den Bus immer für kurze Zeit um kurze Transfers durchzuführen.
- Block mode aka burst mode: Eine Abfolge an Transfers wird auf einmal durchgeführt. Effizienter aber blockiert den Bus über längere Zeit
- Fly-by-mode: DMA teilt dem Gerät mit direkt Daten aus dem Speicher zu lesen/speichern.
- DMA stores/reads word itself: Erlaubt Geräte-zu-Geräte und Speicher-zu-Speicher kopieren

Sicherheit

Ziele

Confidentiality: Daten vor unautorisierten Dritten schützen

Integrity: Daten vor unautorisierter Veränderung schützen

Availability: Sicherstellen, dass niemand das System durch eine DoS-Attacke unnutzbar machen kann

Andere Typen: **Authenticity, Accoutability, Non-Repudiability, Privacy,** …

Adversary Model

Ein Adversary ist im Allgemeinen eine Turing Machine und kann Arbitrary Malicious Operationen ausführen, jedes System sollte ein Adversary/ Threat Model besitzen. Modell beschreibt, welche Möglichkeiten ein Adversary hat.

Dolev-Yao Adversary Model: Das Adversary kann jede Nachricht im Netz abfangen, ist ein autorisierter Teilnehmer im Netzwerk und kann an jeden Teilnehmer senden. Kann Nachrichten von jedem Teilnehmer empfangen und kann Nachrichten unter falscher Identität versenden.

Protection Domain: Domain-Wolken die die Dateien mit den Schreibrechten beinhalten, diese Domain-Wolken können sich schneiden und Schnittmengen bilden.

Protection Matrix: Tabelle(Domain=Zeile, Datei=Spalte, Zelle=Rechte) in der steht, welche Domain welche Rechte auf einer Datei hat.

Access Control Lists (ACLs): Die Prozesse mit Owner sind im User Space, die Dateien mit den Zugehörigen Rechten befinden sich im Kernel Space, diese Liste im Kernel Space ist die ACL, in ihr gibt es zu jeder Datei einen Eintrag in dem steht, welcher User welche Rechte hat.

Authentication

1. Something you know (Password, Pin, …)
2. Something you have (Smartcard, Smartphone, …)
3. Something you are (Biometrische Authentifikation)

UNIX Passwörter

Liste der Nutzernamen und verschlüsselten Passwörtern ist in der passwd Datei. Wenn sich der User einloggt wird das Passwort verschlüsselt und mit dem verschlüsselten Passwort überprüft. Aber anfällig für **Precomputation Attacks:**

1. Angreifer verwendet Wörterbuch mit häufig verwendeten Passwörtern und verschlüsselt diese mit dem bekannten Algorithmus
2. Angreifer nimmt die öffentlich verfügbare Passwort Datei
3. Liste mit Passwörtern in der Datei wird mit den Passwörtern aus dem Wörterbuch verglichen
4. Für jeden Match kennt der Angreifer nun das Plaintext Passwort.

Salted Passwörter

Um gegen Precomputation Attacks besser geschützt zu sein, werden Passwörter gesalted. Jedes Passwort wir mit einem salt (random Nummer) als weiteren Parameter verschlüsselt, der Salt wird als Plaintext mit dem verschlüsselten Passwort gespeichert.

Trusted Computing Base (TCB)

Die TCB besteht aus den Hardware und Software Komponenten die benötigt werden um alle Sicherheitsregeln sicherzustellen.

TCB besteht normalerweise aus: Sicherheitsrelevanter Hardware, Teilen des OS Kernel und Nutzerprogrammen mit Superuser Rechten.

Folgende OS Funktionen müssen Teil des TCB sein: Prozesserstellung und Wechsel, Memory Managment, Part of file und I/O Management

Sichere Designs versuchen den TCB möglichst klein und separat vom OS zu halten. Eine Key-Komponente ist der Reference Monitor, dieser entscheidet, ob sicherheitskritische System calls ausgeführt werden sollen oder nicht.

Formal Security Models

Discretionary Access Control (DAC): System in dem der User über die Zugriffsrechte entscheidet.

Mandatory Access Control (MAC): System fordert zwingende Zugriffsrechte, die vom Nutzer nicht verändert werden können

Bell-LaPadula Security Model

Designed um sensitive Informationen in Organisationen zu verwalten. Nutzer und Objekten werden Security Levels zugeordnet. Zwei Regeln werden auf die Informationsflüsse angewendet:

1. Simple Security Property: Prozesse auf Level k können nur Objekte auf Level k oder niedriger lesen.
2. * Property: Prozesse auf Level k können nur Objekte auf Level k oder höher schreiben.

Diese „read down und write up" Policy stellt sicher, dass keine Informationen einer höheren Schicht an eine niedrigere gelangen können.

Biba Model

Bell-LaPadula Modell ist auf Confidentiality ausgelegt, Biba Modell ist auf Integrity ausgelegt, dafür gibt es folgende Regeln:

1. Simple Integrity Property: Prozesse auf Level k können nur Objekte auf Level k oder niedriger schreiben (no write up)
2. Integrity * Property: Prozess auf Level k kann nur Objekte auf Level k oder höher lesen.

Covert Channels

Prozesse können Informationen an andere Prozesse durch so genannte Covert Channels geben, selbst wenn durch die Security Policies eine direkte Kommunikation unterbunden ist. Informationen werden übergeben indem bestimmte Systemeigenschaften geändert werden, die der Empfänger lesen kann: Modulating CPU Usage, Modulating Paging Rate, Locking Files, Acquiring /releasing decicated Devices, …

Side-Channels

Jede Information, die aus einer Implementation eines Computer Systems gewonnen werden kann: Cache leakage, timing information, power consumption, electromagnetic radiation, …

Malware

Malware steht für malicous Software die Hintertüren in kompromittierten Systemen installieren kann und die Kontrolle über eine Machine hat, diese zu einem „Zombie" macht, eine Sammlung solcher „Zombies" nennt man ein botnet.

Ransomware: z.B. Festplatte verschlüsseln, um Geld zu fordern

Spyware: Sensitive Daten klauen, um z.B. Identitätsdiebstahl zu begehen.

Trojan Horses: Software die nützlich erscheint aber versteckte, schädliche Funktionen enthält. Meistens durch den User runtergeladen und installiert. Ein Trojaner kann dann fast alles auf dem Host.

Viruses: Viren sind Programme, die sich vermehren in dem sie sich an Programme anhängen. Wenn ein Virus auf einem System ist versucht wer sich an andere Programme zu hängen und sein Virus Payload auszuführen.

Worms: Ähnlich zu Viren aber Würmer können sich auch durch das Netzwerk verteilen und andere Systeme befallen

IoT Malware: Angriff auf IoT Devices, da diese schlechte Sicherheitsfunktionen haben. Internet scannen um angreifbare Geräte zu finden.

Virtualisierung

Warum Virtualisierung?

Fault tolerance: Crash eines Systems lässt andere Systeme unberührt

Isolation: Sicherheitsvorfälle auf einem virtuellen System geben nicht automatisch sensitive Informationen über andere Systeme auf

Efficient Resource Use: Keine dedizierte Hardware benötigt, Weniger Energieverbrauch, Weniger benötigter Rack Space, Kostensenkung

Deployability

Entwickeln und ausrollen neuer Apps ist einfacher: Viele Programme brauchen bestimmte OS Versionen, spezielle Libraries, Dateien etc, daher Programme in einer VM ausrollen die alle benötigten Software enthält deutlich einfacher. Dieser Prozess wird „Virtual Appliance“ genannt.

VMs migrieren ist einfacher als einen Prozess zu migrieren: Einfrieren und pausieren von laufenden VMs, an neuem Ort weiterlaufen lassen. So können auch Kritische Informationen einfach mitgereicht werden.

Reliability

Das System im Hypervisor ist ein Single Point of Failure. Die meisten Fehler passieren in komplexen Programmen (wie z.B. dem OS). Die Codebase des Hypervisors ist typischerweise zwei Order of Magnitude kleiner als die des OS. Die Wahrscheinlichkeit von Programmierfehlern im Hypervisor ist dadurch auch deutlich geringer

Voraussetzungen

Safety: Hypervisor sollte volle Kontrolle über die virtualisierten Ressourcen haben.

Fidelity: Verhalten eines Programms auf einer VM sollte identisch zu dem auf echter Hardware sein.

Efficiency: Der meiste Code sollte in der VM laufen ohne Unterbrechungen durch den Hypervisor

Spezielle Instruktionen

Sensitive Instructions: Instruktionen, die verschieden im user und im kernel mode ausgeführt werden

Priviledged Instructions: Instruktionen, die eine Trap verursachen, wenn sie im User mode ausgeführt werden.

Maschine nur virtualisierbar, wenn Sensitive Instructions $\subseteq$ Priviledged Instructions

Virtualisierung Früher vs Heute

Früher: Keine klare Differenzierung zwischen privileged und nonprivileged Instruktionen

Heute: VM Support seit ca 2005. Container werden verwendet um Gäste-OS auszuführen. Spezielle Instruktionen verursachen eine Exception und eine Trap, die den Hypervisor anstoßen. Der Satz der trapped Operations wird mithilfe einer bitmap in Hardware, welche durch den Hypervisor gesetzt wird, kontrolliert. Ermöglicht den Trap-and-Emulate Ansatz.

Trap-and-Emulate

Gäste-OS denkt es läuft in Kernel mode, wird aber im „real" user mode ausgeführt und virtual kernel mode genannt. Wenn das Gäste-OS eine privileged Operation aufruft, wird diese getrappt. Diese Operation wird dann vom Hypervisor kontrolliert, wenn sie vom Virtual User Mode stammt, wird die Instruktion ausgeführt, wenn sie vom Virtual Kernel Mode stammt, wird die Hardware Reaction emuliert.

Binary Translation

Was machen, wenn die Hardware nicht alle sensitiven Operationen trappen kann?

Binary Translation wird genutzt um den Gästesystemcode on the fly zu überschreiben. Problematische Instruktionen werden durch sichere Code Sequenzen ersetzt, die die originale Instruktion emulieren. Binary Translation nutzen, um alle sensitiven aber nicht privilegierten Instruktionen zu ersetzten. Der Hypervisor überschreibt jeden basic block (block von Instruktionen ohne branches) vor der Ausführung. Sensitive Funktionen werden mit Calls zum Hypervisor ersetzt. Finale Branch Instruktion wird auch mit einem Call zum Hypervisor ersetzt, kann so den folgenden Code Block anpassen bevor er diesen ausführt.

Paravirtualisierung

Im Vergleich zu voller Virtualisierung ist Paravirtualisierung nicht transparent zum Gäste-OS. Ein Machine-like Interface bietet hypercalls um sensitive Operationen auszuführen. Erlaubt simplere und schnellere Implementierung von Systemen.

Dateisysteme

Prozesse sind Abstraktionen von CPU Ausführungen. Address Spaces sind Abstraktionen vom physischen Speicher. **Dateien** sind Abstraktionen von Datenblöcken auf der Festplatte. Ein **Dateisystem** kontrolliert, wie Daten auf einer Festplatte gespeichert und geladen werden. Es gibt viele verschieden Arten von Dateisystemen.

Physische Struktur einer HDD

Eine HDD sind mehrere Platten, die zylindrisch angeordnet sind. Auf jeder Oberfläche läuft ein Read/Write-Head (synchron über alle Platten). Eine Platte ist in mehrere Sektoren unterteilt (Wie Pizzastücke) und die Daten werden auf einer Track gespeichert (Ringförmig wie bei einer CD).

Verzögerungen bei einem Festplattenzugriff

Seek time: Zeit um Arm mit Read/Write-Head zum richtigen Track zu bewegen (ca 10ms)

Rotational Delay: Durchschnittliche Zeit, bis der passende Sektor unter den Read/Write-Head gedreht wurde (ca 4ms)

Read/Write time: Zeit benötigt zum lesen/ schreiben (ca 50 Mikrosekunden)

Dateitypen

Reguläre Dateien: Typischerweise genutzt um Nutzerdaten zu speichern

Directories: Systemdateien um die Struktur eines Dateisystems zu speichern

Character special files: Benutzt um I/O-Geräte zu modellieren

Block special files: Benutzt um Festplatten zu modellieren

File Naming
Wenn eine Datei erstellt wird, wird ihr ein Name zugewiesen. Dieser Name erlaubt es anderen Prozessen auf dieselbe Datei zuzugreifen. Namenskonvention ist abhängig vom OS (z.B. Case-sensitivity und ob Dateiendungen benötigt werden).

File Structure
Drei Typen für Dateistrukturen: byte-oriented, record-oriented, tree-structured

Dateizugriffe
Sequential access: Bytes werden nur in sequentieller Reihenfolge gelesen

Random access: Bytes einer Datei „out-of-order“ lesen ist möglich.

Positioning file reads: Jede read-Operation spezifiziert den Ort zum starten oder eine spezielle Seek-Operation wird genutzt um die aktuelle Position in eine Datei zu bewegen.

Dateiattribute
s. VL 9 Folie 29/30

Directories
Dateien werden normalerweise in Directories oder Ordnern organisiert, Directories selbst sind auch Dateien, frühere Systeme hatten nur ein root directory. Hierarchische Directories erlauben mehrere Ebenen von Directories, um Dateien nach user und themaspezifischen Gruppen zu organisieren.

Special Filenames
„.“ Verweist auf das aktuelle Verzeichnis

„..“ verweist auf das Elternverzeichnis des aktuellen Verzeichnis

Diese Shortcuts erlauben es auf Ordner und Eltern-Ordner zuzugreifen ohne den gesamten Pfad angeben zu müssen.

Festplattenorganisation
Festplatten sind in einen **Master Boot Record (MBR)** und eine oder mehr Festplattenpartitionen unterteilt. Der MBR wird in sector 0 von jeder Festplatte gespeichert und verwendet um den Computer zu booten. Der MBR beinhaltet am Ende den Partition Table, in diesem wird immer genau eine Partition als aktiv markiert.

Festplattenpartitionen
Alle Festplattenpartitionen starten mit einem boot block. Das Layout der Partition ist abhängig vom Dateisystem und kann stark variieren, startet aber typischerweise mit einem superblock, der key Parameter enthält, wie z.B. Anzahl der Blöcke, Nummer zum indentifizieren des Dateisystems.

Datei Allokation
Das Dateisystem muss verwalten, wie Blöcke von Dateien auf den physischen Festplattenblöcken verteilt werden.

Allokationstypen sind Contiguous und Linked-List Allocation

Datenstrukturen, die File Allokation implementieren sind File Allocation Table (FAT) und I-Nodes.

Contiguous Allocation
Dateien gespeichert als kontinuierlicher Lauf von Festplattenblöcken. Einfach zu implementieren und sehr effizient, da nur eine Seek-Operation durchgeführt werden muss. Problem mit externer Fragmentierung und die maximale Dateigröße muss vorher bekannt sein.

Linked-List Allocation
Die Festplattenblöcke, welche eine Datei enthalten werden als Linked-List gespeichert. Erstes Wort eines jeden Blocks beinhaltet den Pointer zum nächsten Block. Keine externe Fragmentierung, Directory Eintrag muss nur die Festplattenadresse des ersten Blocks speichern. Random Access ist sehr langsam, Blockgröße ist keine 2-er Potenz.

File Allocation Table (FAT)
Datenstruktur im Hauptspeicher mit Zeigern zu folgenden Dateiblöcken. Directory Eintrag beinhaltet nur den Zeiger zum ersten Block. Blockgröße kann eine 2-er Potenz sein, Random access ist deutlich einfacher aber die gesamte Tabelle muss im Speicher sein und große Festplatten benötigen große Tabellen.

I-Nodes
Datenstruktur die Dateiattribute und Adressen von Festplattenblöcken beinhaltet. I-node muss nur im Speicher sein, wenn die Datei verwendet wird. Braucht weniger Speicher als FAT.

Directory Implementation
Directory ist eine Datei, die Einträge für jede Datei in diesem Ordner beinhaltet. Jeder Eintrag kann entweder ein fixed-Size Eintrag mit Dateiattributen sein oder ein Link zu einem i-node, der die Dateiinformationen beinhaltet.

Shared Files
Dateien können in mehreren Directoires auftreten indem Verweise verwendet werden. Hard Link: Direkter Link zum i-node der Zieldatei. Symbolic Link: Spezielle Datei, die den Dateinamen der Zieldatei beinhaltet.

Journaling File Systems
Bietet Robustheit gegen Systemcrashes, Systemcrashes können ein Dateisystem in einem inkonsistenten Satus lassen. Journaling sorgt dafür, dass alle Operationen beendet sind, selbst bei einem Crash. Zuerst wird die auszuführende Operation in einen Log geschrieben und dieser gespeichert, dann wir die Operation ausgeführt und danach wird der Eintrag wieder aus dem Log entfernt. Bei einem Systemcrash können so die Operationen, die im Log stehen nochmal ausgeführt werden.

Virtual File Systems
Erlaubt mehrere verschiedene Dateisysteme auf dem selben System. Definiert standardisiertes Interface für Dateioperationen, die für Nutzerprozesse und unterliegende Dateisysteme sichtbar sind.

Design Considerations
Block Size: Große Blockgröße verschwendet Speicher durch interne Fragmentierung. Kleine Blockgröße führt zu Dateien, die über viele Blöcke verteilt sind und dadurch eine höhere Latenz haben.

Verwalten von freien Blocks: Linkded-List von freien Festplattenblöcken oder Bitmap.

Disk Quotas: Dateisysteme limitieren, wie viele Dateien (i-nodes) und Festplattenblöcke ein User verwenden kann. Verwendet dafür eine Quota Datei. Wenn ein Nuter eine Datei öffnet, werden die Attribute und Festplattenblockadressen in einer open-files Tabelle gespeichert. Eine zweite Tabelle beinhaltet alle Quota Einträge für jeden Nutzer, welcher Dateien geöffnet hat. Jedes mal wenn eine Datei geöffnet wird, wird ein Link zu dem Quota Eintrag des Dateiowners erstellt. Wenn die Datei mehr Blöcke benötigt wird zunächst der Quota Eintrag überpürft. Softlimit darf temporär überzogen werden. Hardlimit kann nicht überschritten werden und führt beim Versuch zu einem Fehler. Softlimits werden bei jedem User check-in überprüft und beim erreichen des Softlimits wird der Nutzer gewarnt.

Interprocess Communication (IPC)

Wie bereits erwähnt müssen Prozesse miteinander Kommunizieren um Informationen auszutauschen und koordinierte Aktionen auszuführen. Dafür gibt es folgende Methoden: shared memory, message passing und Remote Procedure calls.

Remote Procedure Call (RPC)

RPC ist eine Abstraktion eines Funktionsaufruf zwischen Prozessen in einem Netzwerk. Verwendet Stubs, also client-side proxies für die tatsächliche Funktion auf dem Server. Der client-side stub findet den richtigen Server und ordnet die Call Parameter. Der Server-side stub empfängt die Nachricht, entpackt die geordneten Parameter und führt die Funktion auf dem Server aus.

RPC vs Regular Procedure Calls

Verschiedene Fehler-Semantiken: Bei RPC passieren durch das Netzwerk mehr Fehler. Es muss eine „exactly once“ Semantik implementiert werden: RPC Calls kann durch das Netzwerk Fehler haben, dupliziert oder mehrfach ausgeführt werden. Server-seitig müssen genug Status-Informationen speichern um Duplikate zu eliminieren. Server muss alle RPCs bestätigen. Client muss unbestätigte RPCs neu senden.

Race Conditions

Eine Race Condition ist eine Situation, bei der zwei oder mehr Prozesse geteilte Daten schreiben und lesen. Das Endergebnis ist abhängig davon, wann jeder Prozess seine Operation auf den geteilten Daten ausführt.

Überprüfen lassen sich Race Conditions mithilfe eines Zählers, der bei einem Producer inkrementiert und bei einem Konsumer dekrementiert wird.

Kritische Regionen

Teil eines Programms bei dem Prozesse zugriff auf den geteilten Speicher bekommt oder diesen bearbeitet nennt man kritische Region oder kritische Sektion. Race Conditions können umgangen werden, wenn nur ein Prozess zur Zeit in seiner kritischen Region ist.

Race Conditions verhindern

Voraussetzungen:

- Keine zwei Prozesse dürfen gleichzeitig in deren kritischer Region sein
- Keine Vermutungen über Speed und Anzahl der CPUs aufstellen
- Kein Prozess außerhalb seiner kritischen Region darf andere Regionen blockieren
- Kein Prozess sollte für immer darauf warten in seine kritische Region zu gelangen

Mutual Exclusion

TODO

Jan Schulz-Nigmann
03.04.2023

Mechanismen für Prozesskoordination

Disabling Interrupts

Prozess schaltet Interrupts aus, wenn er in die kritische Region wechselt. Schaltet Interrupts wieder an, kurz bevor er die kritische Region verlässt. Kein andere Prozess darf anlaufen, wenn ein Prozess in der kritischen Region ist.

Ein paar Probleme bestehen: Wie sicherstellen, dass ein Prozess die Interrupts wieder einschaltet? Funktioniert nicht auf Multiprozessor Maschinen.

Lock Variables

Einzelne geteilte **lock** Variable wird für Koordination verwendet. Initial ist lock=0, Wenn ein Prozess in eine kritische Region eintreten möchte, überprüft er lock, wenn lock=0, kann er ein treten und setzt lock=1, danach resettet er lock wieder. Wenn lock=1 wartet der Prozess bis lock=0.

Wenn ein Prozess abgebrochen wird, nachdem er das lock überprüft hat und ein anderer Prozess läuft und in die kritische Region eintritt kann eine Race Condition auftreten.

Process Alternation

Geteilte Variable **turn** wird genutzt, um zu überprüfen welcher Prozess erlaubt ist in die kritische Region zu wechseln. Initial turn=0, jeder Prozess überprüft ob er in seine kritische Region eintreten kann, wenn turn= 0 kann Prozess 0 eintreten, wenn turn=1 kann Prozess 1 eintreten.

Peterson's Solution

Die Variable turn gibt an, welcher Prozess in seine kritische Region wechseln kann. Das flag array **interested** wird benutzt um anzuzeigen, ob ein Prozess in seine kritische Region eintreten möchte. Nur wenn turn=i und die anderen Prozess flags nicht gesetzt sind, darf Prozess i in seine kritische Region eintreten.

Atomic Instructions: TSL

Mache CPUs bieten atomare Instruktionen wie z.B. TSL (Test and Set Lock) um Mutual Exclusion mit Lock Variablen zu realisieren.

TSL RX, LOCK: Lädt Speicherwort an der Adresse Lock in das Register RX, in Lock wird ein nicht-0 Wert gespeichert. Beide Schritte werden automatisch ausgeführt durch Sperren des Memory Bus. Nachdem der Prozess RX geprüft hat, wenn RX=0, ist kein Prozess in einer kritischen Region und der Prozess kann in seine kritische Region eintreten. Wenn RX!=0 muss der Prozess warten, da ein anderer Prozess in einer kritischen Region ist. Beim Verlassen einer kritischen Region wird LOCK wieder auf 0 gesetzt.

Prozess ruft enter_region auf wenn er in eine kritische Region wechseln möchte. Die Funktion ist ein Spin lock, welches wartet solange LOCK!=0. Wenn ein Prozess eine kritische Region verlässt, ruft er leave_region auf, um das Lock zu resetten.

Atomic Instructions: XCHG

Eine andere atomare Instruktion um Mutual Exclusion mit Lock Variablen zu realisieren ist XCHG. Automatisches Tauschen des Inhaltes von zwei Orten.

Sleep and Wakeup

Warten ist nicht optimal, High-Priority Prozesse könnten auf Low-Priority Prozesse warten (Priority Inversions Problem). Alternativ: Prozesse blockieren, wenn diese nicht in ihre kritische Region eintreten können. Verwendet Sleep und Wakeup Calls.

Jan Schulz-Nigmann
03.04.2023

TODO Producer und Consumer

Semaphores
Koordinationsmechanismus von Dijkstra verwendet Wert, der die gespeicherten Wakups zählt, dabei gibt es zwei Operationen, up und down. Diese werden atomar ausgeführt, bis sie vollständig sind oder der Prozess blockiert.

Down überprüft den Semaphoren Wert, wenn Wert > 0 wird der Wert reduziert und es geht weiter. Wenn der Wert = 0, wird der Prozess eingeschläfert und die Down-Operation bleibt in einem pending state.

Up erhöht den Wert und wenn einer oder mehrere Prozesse schlafen, wird ein Prozess aufgeweckt und kann seien Down-Operation ausführen. Im Endeffekt bleibt der Semaphore Wert gleich aber es schläft ein Prozess weniger auf ihm.

Mutexes
Vereinfachte Version eines Semaphores, welche benutzt wird, wenn nur Mutual Exclusion benötigt wird. Zwei Zustände: locked und unlocked. Mutex_lock wird benutzt um in die kritische Zone zu gelangen. Wenn mutex bereits gelocked ist, wird der aufrufende Thread blockiert bis der Thread in der kritischen Region mutex_unblock aufruft. Mutexes können komplet im User Space umgesetzt werden, wenn atomare Instruktionen zur Verfügung stehen. Mutex_trylock gibt die Möglichkeit mutex zu überprüfen ohne zu blockieren.

Condition Variables
Erlaubt Prozessen zu blockieren, bis eine bestimmte Bedingung eingetreten ist. Ist mit Wait und Signal Operationen implementiert. Prozess kann blockieren in dem er wait auf einer Condition Variable ausführt. Andere Prozesse können in ihre kritische Region wechseln und den schlafenden Prozess mit signal aufwecken, wenn die Condition Variable fertig ist. Wenn mehrere Prozesse auf derselben Condition Variable schlafen, wird eine zufällig ausgewählt.

Monitore
Ein Monitor ist eine Sammlung von Funktionen, Variablen und Datenstrukturen um das schreiben von Programmen zu vereinfachen, die Mutual Exclusion benötigen. Monitore sind Programmiersprachenlevel-Konstrukte. Prozess kann Monitore verwenden aber nicht direkt die internen Datenstrukturen erreichen. Es stellt sicher, dass nur ein Prozess im Monitor aktiv sein kann. Der Monitor wird vom Compiler implementiert und meistens mittels Mutex oder Binary Semaphoren realisiert.

Deadlocks
Eine Sammlung von Prozessen ist im Deadlock, wenn jeder Prozess in der Sammlung auf ein Event wartet, welches nur durch einen anderen Prozess in der Sammlung ausgelöst werden kann.

Resourcen
Preemptable resources: Ressourcen die mit Gewalt von einem Prozess entnommen werden können ohne negative Effekte.

Non-preemptable resources: Ressourcen, welche nicht einfach weg genommen können ohne womöglich Fehler zu verursachen.

Resource Access Pattern: Request Resource, Use Resource, Release Resource

Resource Acquisition: Wenn Ressourcen nicht verfügbar sind, muss der Anfrageprozess warten. Der Ressourcenanfrageprozess ist sehr von der OS Implementation abhängig. Ein spezieller Anfrage-System Call oder ein open System call auf eine spezielle Datei erlaubt exklusiven Zugriff für einen Prozess.

Resource Deadlock Voraussetzungen

Alle Voraussetzungen müssen erfüllt sein, damit ein Deadlock auftreten kann.

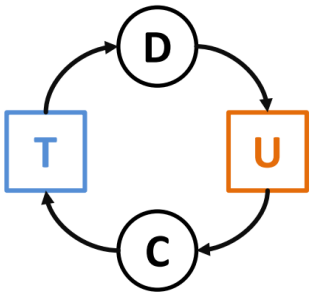
1. Mutual Exclusion: Jede Ressource ist entweder verfügbar oder zu genau einem Prozess zugeordnet
2. Hold and wait: Prozesse, welche gerade Ressourcen halten können weitere Ressourcen anfragen
3. No pre-emption: Zugeordnete Ressourcen können nicht mit Gewalt entfernt werden
4. Circular wait condition: Es muss eine kreisförmige Liste geben in der jeder Prozess auf eine Ressource wartet, die vom nächsten Prozess gehalten wird.

Deadlock Modellierung

Ressourcenallokationsgraphen sind ein gutes Tool um Deadlocks zwischen Prozessen zu identifizieren

Resource R assigned to process A

Process B requesting resource S



Deadlock situation between processes C and D

Ansätze um mit Deadlocks umzugehen

Problem ignorieren (Straus Algorithmus)

Pragmatischer Ansatz: Manchmal macht es keinen Sinn den Aufwand zu betreiben um gegen Deadlocks zu schützen. Ist abhängig davon, wie wahrscheinlich ein Deadlock ist, wenn Deadlocks weniger wahrscheinlich als andere System Crashes sind, lohnt es sich nicht um die Deadlocks zu kümmern. Birgt aber Performance Penalty und ist Unpraktisch.

Deadlock erkennen

Ressourcengraphen aufstellen, und den Graphen auf Kreise untersuchen

Algorithmus: Für jedes Node N im Graphen sollen folgende Schritte mit N als Startknoten ausgeführt werden:

1. L zu einer leeren Liste initialisieren und alle Pfade als unmarkiert markieren.
2. Aktuellen Knoten an das Ende der Liste anhängen und überprüfen, ob der Knoten jetzt 2 mal in L auftaucht, Falls ja gibt es einen Kreis im Graphen und der Algorithmus wird beendet.
3. Wenn es unmarkierte ausgehende Pfade vom aktuellen Knoten gibt, gehe zu 4. Sonst zu 5.
4. Einen zufälligen unmarkierten Pfad auswählen und markieren. Dann dem Pfad folgen und bei Schritt 2. Weiter machen

5. Wenn der aktuelle Knoten der initiale Knoten ist, beinhaltet der Graph keine Kreise, der Algorithmus terminiert. Sonst ist der Knoten eine Sackgasse und der Graph wird zum vorherigen Knoten zurückgetraced: Knoten aus L entfernen und vorherigen Knoten als aktuellen setzen und weiter mit Schritt 3.

Erkennen mit Mehreren Ressourcen: Kompliziertere Buchhaltung wenn mehrere Ressourcen des selben Typs existieren

- E – Existing resource vector: Vektor E $= (E_1, E_2, \dots, E_m)$ spezifiziert die gesamte Anzahl von jeder Ressource in der Klasse i ($i \leq i \leq m$)
- A – Available resource vector: Vektor $A = (A_1, A_2, \dots, A_m)$ spezifiziert die nicht zugewiesene Anzahl an Ressourcen für jede Ressourcenklasse i
- C – Current allocation matrix: Array das beschreibt, wie viele Ressourcen einer Klasse jeder Prozess hält. Eintrag C_{ij} spezifiziert, wie viele Ressourcen von Klass j Process P_i aktuell hält. Es gilt: $\sum_{i=1}^{n} C_{ij} + A_j = E_j$
- R – Request Matrix: Beschreibt, wie viele Ressourcen jeder Klasse ein Prozess benötigt, Eintrag R_{ij} spezifiziert, wie viele Ressourcen der Klasse j Prozess P_i benötigt

Erkennung durch vergleichen der Vektoren und deren Matrizen.

Algorithmus: Zu beginn sind alle Prozesse nicht markiert

1. Suchen nach Prozess P_i für den Zeile i in R kleiner oder gleich zu A ist
2. Wenn so ein Prozess gefunden wird, i-te Zeile von C zu A addieren und Prozess P_i markieren, dann zurück zu Schritt 1.
3. Falls kein solcher Prozess existiert, endet der Algorithmus

Nach Ende des Algorithmus sind alle nicht markierten Prozesse im Deadlock.

Deadlock Recovery

Pre-emption: Ressourcen vom aktuellen Besitzer entfernen und an einen anderen Prozess geben. Nachteil: Nicht alle Ressourcen können einfach weggenommen werden.

Killing Processes: Prozesse töten, die in einem Wartekreis sind, Prozesse töten bis der Kreis gebrochen ist. Alternativ können auch andere Prozesse getötet werden, wenn sie Ressourcen aus dem Deadlock freigeben.

Rollback: Prozesse speichern regelmäßig Checkpoints. Bei einem Deadlock kann der Prozess auf einen Checkpoint zurückgerollt werden indem er noch nicht im Deadlock ist. Nachteile: Die Arbeit nach dem Checkpoint geht verloren, nicht alle Aktionen können ohne Seiteneffekte durchgeführt werden.

Deadlocks Avoidance durch dynamische Ressourcenallokation

System muss entscheiden, wann vergeben einer Ressource „sicher“ ist. Ressourcetrajektorien können dafür verwendet werden.

Safe und Unsafe States: Ein Status ist sicher, wenn es eine scheduling order gibt, bei der jeder Prozess beendet werden kann, auch wenn alle Prozesse sofort zu beginn alle benötigten Ressourcen einfordern.

Banker’s Algorithmus: Algorithmus überprüft ob eine Anfrage zulassen zu einem unsicheren Status führt, Anfrage wird nur im Fall „sicher“ zugelassen.

1. Suchen nach einer Zeile R, bei der die nicht erfüllten Ressourcenanforderungen kleiner oder gleich zu A sind. Wenn keine solche Zeile existiert, wird das System in einen Deadlock geraten.
2. Angenommen ein Prozess fordert alle Ressourcen an und wird dann beendet. Dieser Prozess wird als terminated markiert und seine Ressourcen werden zu A addiert.
3. Wiederholen von Schritt 1 und 2 bis alle Prozesse als terminiert markiert sind (safe state) oder kein Prozess existiert, dessen Ressourcenanfrage erfüllt werden kann (Deadlock)

Deadlocks Prevention durch nicht-erfüllen einer der Voraussetzungen

Probleme mit Deadlock Avoidance: Prozesse wissen selten vorher welche Ressourcen sie benötigen, die Anzahl der Prozesse ist nicht fest und Ressourcen können verschwinden.

Lösung versucht Deadlocks zu verhindern indem eine Voraussetzung für Deadlocks nie erreicht wird.

Mutual Exclusion: Kein Prozess darf exklusiven Zugriff auf eine Ressource haben, bei Daten durch read-only, meist aber nicht möglich umzusetzten. Deamon-Prozesse: Prozesse greifen nie direkt auf Ressourcen zu, sondern über deamon Prozesse, welcher nie andere Ressourcen anfragt.

Hold-and-Wait: Verhindern eines Prozesses, Ressourcen zu halten, während dieser auf andere Ressourcen wartet. Alle Prozesse müssen alle Ressourcen bei Beginn anfragen. Prozess wird nur ausgeführt, wenn alle Ressourcen vorhanden sind und er bis zum Ende laufen kann. Probleme: Ressourcenbenutzung ist vorher nicht bekannt und Ressourcennutzung möglicherweise nicht optimal.

No-Pre-Emption: Ressourcen von Prozessen mit Gewalt entfernen, ist aber bei machen Ressourcen schwierig -> Ressourcen virtualisieren, ist aber auch nicht für alle Ressourcen möglich.

Circular Wait: Beschränkungen auf Ressourcennutzung eines Prozesses, jeder Prozess darf nur eine Ressource zur Zeit halten, so kein simultaner Zugriff auf mehrere Ressourcen möglich. Globale Nummerierung für alle Ressourcen, alle Ressourcenanfrage müssen in numerischer Folge passieren, verhindert so Kreise.

Two-Phase Locking

In der ersten Phase beantragt der Prozess alle Ressourcen, die er benötigt um eine Operation auszuführen. Wenn alle Ressourcen erhalten wurden, startet die zweite Phase: Operation durchführen und Ressourcen wieder freigeben. Wenn nicht alle Ressourcen in der ersten Phase erhalten werden können, werden alle Ressourcen wieder freigegeben und der Prozess startet Phase Eins erneut. Kann nicht in realtime und process-control Systemen angewandt werden.

Communication Deadlocks

Zwei Prozesse, A und B, kommunizieren per Nachricht. A sendet Anfragenachricht und blockiert bis er eine Antwort von B erhalten hat, B ist blockiert bis er die Anfragenachricht von A bekommt und antwortet dann. Wenn die Anfragenachricht verloren geht, sind beide Prozesse für immer blockiert. Dies nennt man einen Communication Deadlock, Lösung: Timeout für Deadlock Detection und Wiederherstellen (durch z.B. Nachricht erneut senden)

Livelock

Bezeichnet eine Situation in der zwei Prozesse jeweils eine Ressource halten und eine weitere Ressource benötigen, die der jeweils andere Prozess gerade besitzt. Ressourcen freigeben und erneut versuchen. Wenn das synchron passiert, blockieren sich die Prozesse weiterhin gegenseitig, keiner der beiden Prozesse kann weiterlaufen auch wenn sie sich nicht direkt blockieren

Starvation

Entsteht durch die Ressourcen Allocation Policy, welche entscheidet, welche Ressourcen an welche Anfrage gegeben werden. Starvation beschreibt, dass Prozesse für immer auf ihre Ressourcen warten müssen, da andere Prozesse die Ressource immer zuerst bekommen. Kann verhindert werden indem man First-Come-First-Served einsetzt.