- **TLB**: Translation Lookaside Buffer für Daten, Code und Heap (am Anfang des Stacks); CPU -> TLB -> ...
- **PMLE -> PDPE -> PDE -> PTE -> Page**
- **32-Bit VA (4 KiB)**: 20 Bit Page, 12 Bit Offset
- **32-Bit VA (4 MiB)**: 10 Bit Page, 22 Bit Offset

Virtuelle Adressierung

In einem 32-Bit-System mit `4kB` großen Seiten wird die zweistufige virtuelle Adressierung eingesetzt.

Wie viele Speicher und wie viele Tabellen braucht man mindestens für die Speichertabellen (PD, PT) eines kleinen Programms (mit Code, Date, Heap und Stack)?

```
2 PT, 1 PD (kleines Programm)

32 Bit -> 10 PD, 10 PT, 12 Offset

2^10 = 1024 Eintraege in dem PD
Jeder der Eintraege hat 32 Bit

=> Insgesamt 1024 * 32 Bit = 4 KiB

2^10 = 1024 Eintraege in der PT (= 4 KiB)

=> Insgesamt 4 KiB (PD) + 2 * 4 KiB (PT) = 12 KiB fuer d

Falls fuer Code noch Speicher, dann noch `4 * 4 KiB` (4 
evtl. `2 * 4` KiB falls Daten, Code und Heap in ein Page
```

`=> Insgesamt Insgesamt 28 KiB`

Wie groß sind die Register?

- 2^9 Einträge pro Tabelle
- (64 Bit): 2^9 _ 64 Bit = 512 _ 8 Bit = 4 KiB

Warum ist diese Tabellengröße gewählt? Was kann man tun, um größere Seiten hinzubekommen?

Optimum aus Zugriffzeiten bei Miss und Speicherbedarf. Bei invalidem Eintrag (P = 0) kann man sich die folgenden Tabellen sparen. Um groessere Seiten hinzubekommen muss man die Tabellen verkleinern und den Offset vergroessern. Fuer 4 MB Seiten zum Beispiel 22 Bit Offset, fuer 4 KB nur 12 Bit Offset.

Warum wird die einstufige virtuelle Adressierung nicht verwendet?

Auch fuer kleine Anwendungen muss die komplette Tabelle erstellt werden. Dadurch gibt es einen sehr hohen Speicheraufwand.

Wie arbeitet die virtuelle Adressierung tatsächlich?

In echt verwendet: Mehrstufige virtuelle Adressierung.

Die Virtuelle Adresse wird in mehrere Teile aufgeteilt und durch mehrere Tabellenstufen geleitet, um die physische Adresse zu bestimmen. Der Index jeder Tabelle wird verwendet, um den Eintrag in der nächsten Tabelle zu finden, bis schließlich die physische Adresse erreicht wird (anhand von Kombination mit Offset). Dies ermöglicht eine effiziente Nutzung des virtuellen Adressraums und eine flexiblere Speicherverwaltung.

Dateisysteme

ext2

- ext2 ist ein Linux-Dateisystem, das auf dem älteren ext-Dateisystem basiert.
- (+) Es verwendet Inodes zur Verwaltung von Dateien und Verzeichnissen.
- (+) Die Dateien werden in Datenblöcken auf der Festplatte gespeichert.
- (+) Es unterstützt Dateinamen mit bis zu 255 Zeichen und Pfadlängen von bis zu 4096 Zeichen.
- Es bietet grundlegende Funktionen wie Lese- und Schreibzugriff auf Dateien und Verzeichnisse.

  Gegeben sei ext2-Dateisystem mit Blockgröße von ‘4 KiB“.
  Dateikopf: 12 Einträge für direkte Adressierung von Datenblöcken und 3 Einträge für Verweise auf 1 bis 3-fach indizierte Blöcke.

```
BS = 1 KiB
BN = 32 Bit (immer bei ext2)

Anzahl Eintraege indirekte Bloecke = BS (in Bit) / BN =

Max. Groesse = Direkte * BS + Anzahl Eintraege indirekte
Max. Groesse = 12 * 1 KiB + 256 * 1 KiB + 256^2 * 1 KiB
```

ext4

- ext4 ist ein Linux-Dateisystem, das auf ext2 basiert.
- Es bietet erweiterte Funktionen wie Journaling, erweiterte Zugriffskontrolle und erweiterte Zeitstempelung.

Aufbau

- Besteht aus Bootsektor, Superblock, Inodes, Datenblöcken und Journal.
- Bootsektor enthält Startcode und Informationen über die Partition.
- Superblock enthält wichtige Informationen über Dateisystem (Größe der Partition, Blockgröße & Anzahl Inodes)
- Inodes speichern Metadaten von Dateien und Verzeichnissen, wie Dateigröße, Berechtigungen und Zugriffszeiten.
- Datenblöcke enthalten die tatsächlichen Dateidaten.
- Journal protokolliert Änderungen (Datenkonsistenz, Wiederherstellung nach Systemabsturz)

Funktionsweise mit Extents

- Ext4 verwendet Extents, um die Dateispeicherung effizienter zu machen.
- Extent ist zusammenhängender Bereich von Blöcken auf Festplatte, der Datei repräsentiert
- Anstatt einzelne Blöcke für jeden Teil einer Datei zu verketten, werden die Daten in aufeinanderfolgenden Extents gespeichert
- Dadurch wird Anzahl der Zugriffe auf Festplatte reduziert und Leistung verbessert
- Ext4 verwendet Baumstruktur variabler Tiefe, um Extents zu verwalten und auf sie zuzugreifen.

Größere Partitionen mit Extents

- Durch Verwendung von Extents kann größere Partitionen verwalten, da effizienter mit Speicherplatz umgehen

- Ohne Extents müsste ext4 jeden Block einer Datei einzeln verketten (höheren Overhead)
- Mit Extents werden zusammenhängende Bereiche von Blöcken genutzt, um Dateien zu repräsentieren, was die Verwaltung großer Partitionen erleichtert
- Nutzung von Extents reduziert den Speicherplatzbedarf für Verwaltung der Dateistruktur und ermöglicht mehr Speicherplatz für eigentliche Dateidaten
- Dadurch können größere Partitionen effizienter genutzt werden.

Rechnung größe Datei

```
Pro Datei = 2^32 Bloecke (32 Bit immer bei ext4)
BS = 4 KiB
Max. Datei = Anz. Bloecke * BS
           = 2^32 * 4 KiB = 16 TB
```

FAT (File Allocation Table, MS-DOS)

- Verkettungsmethode, bei der die Dateien in aufeinanderfolgenden Blöcken auf dem Datenträger gespeichert werden.
- Verknüpfungsinformationen in FAT-Tabelle gespeichert, die den Zugriff auf die Datenblöcke ermöglicht.
- kurze Dateinamen im 8.3-Format (NAME.EXT)
- einfacher aufgebaut als ext2 und bietet weniger erweiterte Funktionen

Prozesskommunikation

Wie kann der Empfänger vom Senden der Nachricht informiert werden, wenn er nicht dauernd nachfragen will?

- **Signale**: Sendende Prozess sendet Signal an empfangenden Prozess, um ihn über eine neue Nachricht zu informieren. Empfangende Prozess hat Signal-Handler, der Signal abfängt und Nachricht verarbeitet
- **Systemaufrufe**: Sendende Prozess verwendet spezifischen Systemaufruf, um empfangenden Prozess über Nachricht zu benachrichtigen. Empfangende Prozess ruft entsprechenden Systemaufruf auf, um Nachricht abzurufen
- **Synchronisationsmechanismen**: Synchronisationsmechanismen wie Semaphore, Mutex oder Bedingungsvariablen werden eingesetzt. Sendende Prozess setzt entsprechenden Synchronisationspunkt, um empfangenden Prozess zu benachrichtigen. Empfangende Prozess überprüft Synchronisationspunkt und liest die, falls verfügbar

  Prozess möchte Nachrichtenblöcke variabler Länge schicken

Verwendung einer Nachrichtenwarteschlange (**Message Queue**).

- ermöglicht die Übertragung von Nachrichten zwischen Prozessen.
- Nachrichten können variable Längen haben.
- Nachrichtenwarteschlange funktioniert als Pufferstruktur.
- Der sendende Prozess schreibt Nachrichtenblöcke in den Puffer.
- Der empfangende Prozess liest die Nachrichtenblöcke aus dem Puffer.
- Puffer kann verschiedene Längen von Nachrichten aufnehmen.

  Pipes vs. Message-Queues vs. Signale

Pipes:

- Unnamed und Named Pipes

  - Unnamed entstehen bei Fork zwischen Vater und Kind Prozessen
  - Named koennen zwischen allen Prozessen enstehen (Verwandschaftsgrad egal)
- Unidirektionale Kommunikation
- FIFO Datenfluss
- Verbindung zwischen verwandten Prozessen
- Implementiert mit Dateideskriptoren
- Begrenzter Puffer

Message-Queues:

- Bidirektionale Kommunikation
- Nicht-FIFO oder Prioritäten beim Lesen
- Verbindung zwischen beliebigen Prozessen
- Implementiert als Kernel-Objekte
- Flexible Puffergröße

Signale:

- Ereignisse oder Benachrichtigungen von Prozess gesendet um auf bestimmte Ereignisse hinzuweisen oder bestimmte Aktionen auszulösen
- Diesnen der Kommuniktion und Koordination zw. Prozessen oder zw. Betriebssystem und Prozess

  Wie werden sie behandelt?

- Prozess kann Signale empfangen und darauf reagieren indem er Signal-Handler-Funktionen definiert
- Prozess kann Signal-Handler registeren, um auf bestimmte Signale zu reagieren und entsprechende Aktionen auszuführen
- Verschiedene Möglichkeiten Signale zu behandeln: ignorieren,

ausführen einer Std. Aktion, Ausführen einer benutzerdefinierten Aktion
- Können blockiert werden, um Ausführung vorübergehend zu verhindern
- Können an andere Prozesse gesendet werden, um mit ihnen zu kommunizieren

Warum kann man einzelne Signale nicht überladen?

- Können nicht überladen werden, da sie standardisierte Bedeutungen haben, die konsistent und interoperabel sind
- Einheitliche Signalbehandlung erleichtert Systemimplementierung, -leistung und -vorhersagbarkeit

Warum kann man Signale überladen?

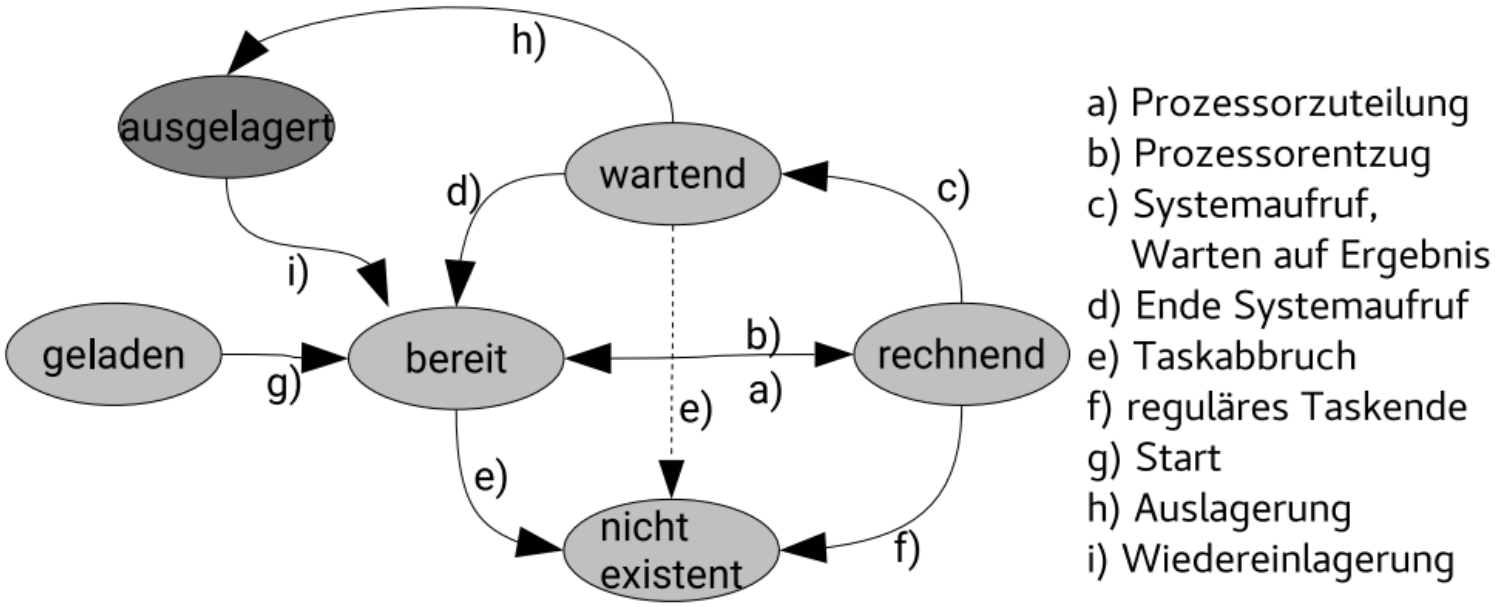
- Signale können maskiert werden, um vorübergehend Behandlung zu unterdrücken
- Maskierte Signale werden von einem Prozess ignoriert und nicht verarbeitet
- Maskierte Signale werden in einer Signalmaske gespeichert, die vom Betriebssystem verwaltet wird
- Das Maskieren von Signalen ermöglicht die Steuerung der Signalverarbeitung auf granularer Ebene
- Maskierte Signale können später wieder entsperrt werden und werden dann normal verarbeitet

Semaphoren:

- Ein Semaphore ist eine Synchronisationsprimitive.
- Es ist ein ganzzahliger Zähler im Kernel.
- werden verwendet, um den Zugriff auf gemeinsame Ressourcen zu steuern.

- helfen bei Synchronisation und Koordination von Prozessen/Threac

Prozesse



Multilevel-Feedback-Scheduling:

- Dialogorientierte Prozesse benötigen wenig Rechenzeit, fordern jedoch kurze Antwortzeiten.
- Beim Start erhält Prozess höchste Priorität
- Wenn Zeitscheibe im nächsten Zyklus vollständig ausgenutzt: tiefere Prioritätsebene
- Wenn Zeitscheibe nicht vollständig ausnutzt: Startpriorität
- Gründe/Vorteile: - Priorisierung von dialogorientierten Prozessen, - Effiziente Ressourcennutzung, - Versetzung rechenintensiver Prozesse in tiefere Prioritätsebenen, - Ausgewogenere Auslastung des Systems

Thread:

- Ausführbare Einheit innerhalb eines Prozesses
- Ermöglichen parallele Ausführung mehrerer Aufgaben innerhalb eines Prozesses

- Teilen sich den Speicherbereich mit anderen Threads desselben Prozesses
- Können auf globale Daten und Variablen des Prozesses zugreifen
- Können unabhängige Aufgaben ausführen und den Prozessor freiwillig abgeben
- **vs. Prozesse**: - Prozess: eigenständige Ausführungseinheiten, Thread innerhalb eines Prozesses, - Prozesse eigenen Adressraum, - Prozesse sind isoliert. - Prozesse erfordern Kontextumstellung, - Threads weniger Overhead, weniger Ressourecen, - Threads parallel
- **Kernel Level Threads**: - KLTs vom BS auf Kernebene verwaltet, - Erzeugen/Verwalten von KLT erfordern Syscalls, - KLTs können blockierende Syscalls verwenden, - BS gewählt KLTs Prozessorzeit basierend auf Sched. Strat., - **Mehrprozessorsystem**
- **User Level Threads**: - ULTs auf Anwendungseben durch Thread-Bibliothek, teil desselbene Prozesses (teilen Adressraum), - erfordern keine Syscalls, - geben CPU-Res. frewillig ab (yield), - BS betrachtet ULTs als einzigen Prozess
- **Speicherbelegung**: (links und rechts "Thread 1"): [Daten | Code | Heap | >Frei< | Stack (to Stack)], (mitte "reeller Speicher"): Daten | Code | Heap | Stack | Stack

Memory Mapped Files:

- Abbildung von Daten in Arbeitsspeicher (auf Inhalt direkt über Speicherzugriffe ohne expl. Lese- und Schreibeoperationen)
- `mmap()` bestimmter Ausschnitt in virt. Adressraum eines Prozesses abgebildet
- Bei Zugriff auf abgebildeten Speicherbereich wird Page Fault ausgelöst, da Daten ggf. nicht im HS vorhanden

- BS lädt enstsprechende Seite der Datei in HS und setzt Anwendung mit fehlgeschlagenen Befehl fort
- Prozess kann nun direkt auf Inhalt d. Datei über Speicherzugriffe zugreifen
- Änderungen im Speicher automatisch in Datei geschrieben, lesen liefer aktuellen Inhalt
- => MMF ermöglicht effizienten Zugriff auf (große) Dateien, vereinfachen Umgang mit Dateiinhalten

Copy-on-Write:

Warum führt `copy_on_write` zur beschleunigten Ausführung eines Child-Prozesses nach `fork()`?

- Copy-on-Write-Strategie beschleunigt Ausführung von Child-Prozessen nach `fork()`.
- Prozesse teilen sich anfangs gleichen Anwendungsadressraum über Pagetables.
- Pagetable-Einträge sind schreibgeschützt `(W = 0)`.
- Schreibzugriff führt zu einer Exception und Kopie der Seite.
- Schreibschutz wird für entsprechende Einträge aufgehoben.
- Effiziente Nutzung des Speichers durch Kopien nur bei Schreibzugriffen.
- Beschleunigte Ausführung des Child-Prozesses.

Was geschieht bei einem Unterprogrammaufruf?

- Ausführung von `exec()` überschreibt den kompletten Adressraum des Kindprozesses.
- Neuer Adressraum wird entweder vollständig oder durch Demand Paging bei Bedarf neu gebildet.

Fünf Aufträge warten auf ihren Start. Ihre erwarteten

Laufzeiten sind 9, 5, 3 und 5 sec. In welcher Reihenfolge sollten sie laufen, damit die mittlere Antwortzeit minimiert wird?

Von der kuerzesten Zeit zur laengsten Zeit, also `3`, `5`, `5`, `9`. Mittlere Antwortzeit ist `(3 + 8 + 13 + 22) / 4 = 11.5`

Cache

Warum werden Seiten ausgelagert?

Um Speicherplatz im Hauptspeicher freizugeben und Platz für andere Seiten oder Prozesse zu schaffen, wenn verfügbare Hauptspeicher nicht ausreicht, um alle aktiven Seiten aller laufenden Prozesse zu halten.

Was geschieht dabei?

Inhalt dieser Seite wird vom Hauptspeicher auf Festplatte oder anderes sekundäres Speichermedium verschoben. Dadurch wird Speicherplatz im HS freigegeben, und Seite wird als "ausgelagert" markiert. Wenn Seite später wieder benötigt, kann sie vom sekundären Speicher in HS zurückgeladen werden.

Welche Strategien gibt es?

- **Optimal**: nur theoretisch
- **FIFO**: Beim Einlagern mit TS versehen, Seiten mit frühestem TS werden zuerst verdrängt, sinnlos, da System-Dienste früh gestartet werden und immer gebraucht werden
- **Clock-Algorithmus**: Zwei Zeiger durchlaufen hintereinander mit etwas Abstand die Pages im RAM. Vorderer Zeiger setzt Access-Bit jeder Seite auf `0`. Wenn auf eine Seite zugegriffen wird, wird dieses Bit auf `1` gesetzt. Der hintere Zeiger prueft, ob

das Access-Bit auf `0` gesetzt ist, ist dies der Fall, wird die Seite ausgelagert.

- **LRU**: Für jede Seite wird Bit-Vektor gespeichert. Direkt nach der Einlagerung sind alle Bits `0`. In regelmäßigen Zeitabständen prüft das BS für jede Page im HS ihr Access-Bit und schiebt es an den Anfang des Bit-Vektors, so dass sich die anderen Bits um eine Stelle verschieben und ein Bit hausfällt. Anschließend setzt er das Access-Bit auf `0`. Sieht man den Bit-Vektor als Binärzahl an, gibt diese Zahl eine Aussage darüber, wie oft in letzter Zeit auf diese Page zugegriffen wurde. Pages mit niedrigerer Zahl können also eher ausgelagert werden.

Was ist ein "Seitenfehler"?

Page Fault tritt auf, wenn Prozess auf Seite im virt. Speicher zugreifen möchte, die sich nicht im Hauptspeicher befindet.

Wodurch wird er verursacht?

- Angeforderte Seite wurde noch nicht in HS geladen, (noch nicht benötigt, od. zuvor ausgelagert)
- Angeforderte Seite wurde ausgelagert, um Platz für andere Seiten oder Prozesse zu schaffen.
- Fehler im Adressraum des Prozesses (=> auf ungültige Adressen zugegriffen)

Welche grundsätzlichen Reaktionen darauf gibt es?

- **Laden**: Von sek. Speicher in HS laden, damit der Prozess zugreifen kann
- **Austauschen**: Wenn der HS voll, muss andere Seite ausgelagert werden. Ausgelagerte Seite wird auf Festplatte verschoben, und angeforderte Seite wird an ihre Stelle in HS geladen.

- **Fehlerbehandlung**: Wenn Seitenfehler auf ungültigen Zugriff oder anderen schwerwiegenden Fehler hinweist, kann BS den Prozess beenden oder eine entsprechende Fehlerbehandlung durchführen

Virtualisierung

Typ-1 Hypervisor (Bare-Metal Hypervisor):

- Das Gastsystem laeuft in einer niedrigen Privilegstufte
- Kritische oder sensitive Instruktionen werfen eine Exception.
- Werden von der VMM (Virtual Machine Monitor) abgearbeitet.
- Wird auch als "Bare-Metal Hypervisor" bezeichnet.
- Wird direkt auf der physischen Hardware (ohne ein Host-Betriebssystem) installiert.
- Hat direkten Zugriff auf die Hardware-Ressourcen des Systems.
- Bietet eine hohe Leistung und Effizienz, da kein Host-Betriebssystem vorhanden ist.

Voraussetzung:

- physische Hardware
- Zugriff auf grundlegenden Hardware-Ressourcen des Systems (CPU, Speicher, Festplatten, Netzwerkgeräte)
- Hardware sollte Virt.tech. unterstützen, z. B. Intel VT-x, um die Virtualisierungseffizienz zu verbessern.
- Hypervisor benötigt möglicherweise spezifische Treiber für Hardware-Komponenten

Gastsystem:

- Privilegstufe der Gastsysteme wird heruntergesetzt
- VMM Schicht, die direkten Zugriff auf Hardware ermöglicht

- Hypervisor kontrolliert Zugriff auf physischen Ressourcen
- HY Stellt virtuelle Umgebung für Gastsysteme bereit
- Durch Isolierung der Gastsysteme voneinander und vom Hypervisor

Typ-2 Hypervisor (Hosted Hypervisor):

- Wird auch als "Hosted Hypervisor" bezeichnet.
- Wird auf einem Host-Betriebssystem installiert.
- Teilt die Hardware-Ressourcen des Host-Betriebssystems mit den virtualisierten Gästen.
- Hat weniger direkten Zugriff auf die Hardware als ein Typ-1 Hypervisor.
- Bietet eine höhere Flexibilität, da das Host-Betriebssystem verschiedene Anwendungen ausführen kann.
- Wird häufig auf Desktop-Computern für Virtualisierungszwecke verwendet.

Voraussetzung:

- Host-Betriebssystem auf physischen Hardware
- Host-Betriebssystem sollte für Virtualisierung geeignet sein und Installation/Ausführung von Hypervisor-Software ermöglichen
- Hypervisor nutzt Ressourcen des Host-Betriebssystems, einschließlich der Hardware-Ressourcen wie CPU, Speicher und Netzwerk.
- Hardware des Systems sollte Virt.tech. unterstützen, um die Virtualisierungsleistung zu verbessern.
- Typ-2 Hypervisor kann von Funktionen und Treibern des Host-Betriebssystems abhängig sein, um eine optimale Funktionalität zu bieten.

Gastsystem:

- Greift nicht direkt auf physische Hardware zu, sondern nutzt Treiber und Schnittstellen des Host-BS
- HY arbeitet als Schicht zwischen dem Host-Betriebssystem und den Gastsystemen
- Host-BS kontrolliert Zugriff auf Hardware-Ressourcen und stellt sie dem Hypervisor zur Verfügung
- HY ermöglicht Ausführung mehrerer Gastsysteme als separate virtuelle Maschinen
- Durch sicheren Zugriff auf die physischen Ressourcen durch den Hypervisor werden die Gastsysteme voneinander isoliert und geschützt

  Welchen zusätzlichen Hardwarebaustein braucht man, um Nested Page Tables zu unterstützten?

MMU-VM

  Was ist ein Container?

- Bieten Anwendungen abgeschottete Umgebung, die unabhängig von anderen Containern ist
- Teilen gleiches Betriebssystem, ermöglichen aber Einbindung unterschiedlicher Versionen von Bibliotheken
- Bestehen aus Images, die Inhalt des Containers sowie Startparameter enthalten
- Engine für Starten, Stoppen und Verwalten der Container zuständig
- Engine ermöglicht Kommunikation zw. Containern sowie mit Anwendungen außerhalb der Container und des Host-Rechners

  Was ist ein Emulator?

- Ermöglicht Nachbildung beliebiger Systeme
- Liest Binärcode des emulierten Systems aus und führt ihn Befehl für Befehl in Softwaremodell des Prozessors aus
- Eignen sich für Entwicklung und Testzwecke
- Geschwindigkeit im Vergleich zum Originalsystem deutlich geringer (20-30% der Leistung)
- Wine spezieller Fall: stellt virtuelle BS-Schnittstelle bereit, um Windows-Anwendungen auf Linux-Systemen

Was ist I/O-MMU?

- Ermöglicht direkten Zugriff von Gastbetriebssystemen auf virtuelle Geräte
- Direkter Zugriff kann Gefahren mit sich bringen, insbesondere bei DMA-fähigen Geräten
  - erlaubt Geräten, Daten direkt zw. Arbeitsspeicher und dem Gerät zu übertragen, ohne CPU zu belasten
- Verwendet Seitentabellen, um Adressen von virt. Geräten auf physische Adressen abzubilden
- VMM konfiguriert die I/O-MMU für Zuordnung von virtuellen Geräteadressen zu physischen AS-Adressen
- Ermöglicht sicheren und effizienten Zugriff der Gastsysteme auf virtuelle Geräte.

Systemaufrufe

Erklären Sie die Funktion und Wirkungsweise folgender System-Calls:

- `fork()`: Erzeugt neuen Prozess, exakte Kopie des aufrufenden Prozesses. Neue Kindprozess erhält eindeutige PID und Kopie des Adressraums, Dateideskriptoren und andere Ressourcen des

Elternprozesses

- `wait()`: Blockiert Elternprozess und wartet auf Beendigung eines / mehrerer Kindprozesse. Aufruf von `wait()` ermöglicht Elternprozess, auf Abschluss der Ausführung seiner Kindprozesse zu warten, bevor er selbst fortfährt.
- `exit()`: Beendet aktuellen Prozess und gibt Ressourcen frei. (Opt.: Beendigungsstatus)
- `exec()`: Startet neues Programm in Prozess, indem aktuelle Programmcode durch Code des neuen Programms ersetzt wird

Beschreiben Sie den Ablauf eines Systemcalls am Beispiel des Aufrufs write() in eine Datei.

- System-Calls durch Interrupts realisiert
- Anwendung wechselt mittels Software-Interrupt in den BS-Kontext
- Auftrag wird ausgewertet und an Treiber weitergeleitet
- Treiber wandelt abstrakten Auftrag in geräteverständlichen Auftrag um
- Treiber gibt CPU frei
- Gerät sendet Interrupt an Treiber nach Abschluss der Auswertung
- Treiber informiert Scheduler für weitere Ausführung
- Interruptreaktionsroutine endet durch Iret

Treiber

Welche Aufgaben hat ein Gerätetreiber?

- Bereitstellung einer allgemeinen Geräteschnittstelle
- Abstraktion der Gerätefunktionen (auf read, write, ...)

Was ist ein Device-Switch-Table?

- Device-Switch-Table ist eine Datenstruktur im Betriebssystem.
- Sie verwaltet den Zugriff auf verschiedene Geräte (Hardware oder virtuelle Geräte).
- Jedes Gerät hat einen Eintrag in der Tabelle.
- Die Tabelle enthält Informationen und Funktionen zur Steuerung und Kommunikation mit den Geräten.
- Sie abstrahiert die Gerätekommunikation und bietet eine einheitliche Schnittstelle.
- Betriebssystem kann auf die Funktionen in der Tabelle zugreifen, um Geräteaktionen auszuführen.
- Sie ermöglicht einfache Erweiterung des Systems um neue Geräte.
- Wichtiger Bestandteil des Gerätetreiber-Frameworks.

Was bedeutet Minor- und Major-Number?

- **Major-Number**: identifiziert den Gerätetreiber, der ein bestimmtes Gerät in einem Betriebssystem unterstützt. Sie stellt eine eindeutige Kennung für den Treiber dar und wird verwendet, um den entsprechenden Treiber im Kernel zu identifizieren.
- **Minor-Number**: identifiziert spezifisches Gerät, das von einem bestimmten Treiber unterstützt wird. Ermöglicht Unterscheidung zwischen verschiedenen Geräten desselben Treibers. Sie kann verwendet werden, um auf bestimmte Geräteinstanzen oder Funktionen innerhalb eines Gerätetreiberbereichs zuzugreifen.

Zusammen bilden die Major- und Minor-Numbers eine eindeutige Kennung für ein Gerät in einem Unix-System.

Was sind Block-Devices?

- Block-Geräte sind Geräte, die Daten in Blöcken (festgelegte Datenmengen) verarbeiten.

- Sie ermöglichen den zufälligen Zugriff auf Datenblöcke und unterstützen Lese- und Schreibvorgänge in größeren Einheiten.
- Typische Beispiele für Block-Geräte sind Festplatten, Solid-State-Laufwerke (SSDs) und CD/DVD-Laufwerke.
- Block-Geräte werden normalerweise zur Speicherung und Übertragung großer Datenmengen verwendet.

  Character Devices?

- Character-Geräte sind Geräte, die Daten zeichenweise verarbeiten.
- Sie ermöglichen den sequenziellen Zugriff auf Daten, d.h. die Daten werden in der Reihenfolge ihres Eintreffens verarbeitet.
- Typische Beispiele für Character-Geräte sind Tastaturen, Mäuse, serielle Schnittstellen und Soundkarten.
- Character-Geräte werden oft für die Eingabe und Ausgabe von Daten verwendet, die in Echtzeit erfolgen oder eine zeichenweise Verarbeitung erfordern.

  Was versteht man unter “raw” und “cooked”?

Zeichenorientierte Geräte können über diese beiden Schnittstellen angesprochen werden. Die “cooked”-Schnittstelle wird verwendet, um z.B. ganze Eingabezeilen einzulesen. Die Benutzereingaben werden dabei aufbereitet (cook) und Zeichen wie Backspace interpretiert, anstatt sie direkt weiterzugeben. Die “raw”-Schnittstelle gibt alle Zeichen, so wie sie vom Nutzer eingegeben werden, an das System weiter, ohne sie zu interpretieren.

Shared Memory

Wie wird Shared Memory realisiert?

- Reservieren eines gemeinsamen Speicherbereichs im Hauptspeicher des Computers
- Zuweisen dieses Speicherbereichs an mehrere Prozesse
- Prozesse können dann direkt auf den gemeinsamen Speicherbereich zugreifen und Daten austauschen, ohne Daten zwischen den Prozessen kopieren zu müssen.

Wozu wird es verwendet?

- Effizienten und schnellen Datenaustausch zwischen Prozessen
- Gemeinsame Nutzung großer Datenmengen ohne Datenkopien
- Kommunikation und Synchronisation zwischen Prozessen in Multithreaded- und Mehrprozessorsystemen
- Implementierung von IPC (Inter-Process Communication)-Mechanismen wie Pipes, Message Queues und Semaphoren.