Fachhochschule Nordwestschweiz
Hochschule für Technik

Modul Betriebssysteme (bsys-iC)

Feedback aus der Hausaufgabe

- Was ist Ihnen aufgefallen?
- Gab es grundlegende neue Erkenntnisse?
- Was hat gefehlt?
- Wieviel Zeit haben Sie aufgewendet?

Lektion 11: Synchronisation, Nebenläufigkeit und verteilte Systeme

Inhalt

- Nebenläufige Prozesse und quasi-gleichzeitige Ausführung
- Auswirkung des Scheduling auf Nebenläufigkeit
- Preemption, gegenseitige Synchronisation, gegenseitiger Ausschluss (Semaphore, Locks etc)
- Reader/Writer Probleme und Lösungen
- Deadlock Erkennung und Vermeidung/Behebung

Motivation

Parallelität - Nebenläufigkeit

- Falls ein Rechner mehrere Prozesse parallel verarbeiten kann (Multi-Tasking), so führt er jeden Prozess als separate Aktivität aus. Diese Prozesse können unabhängig voneinander ablaufen (Nebenläufigkeit), müssen jedoch synchronisiert werden, da bestimmte Ressourcen (insbesondere die CPU) gemeinsam genutzt werden → Quasi-Parallelität.
- Stehen mehrere CPU‘s zur Verfügung, können Prozesse parallel ablaufen, jedoch weiterhin mit Synchronisation und Wartezuständen bei gemeinsam genutzten Ressourcen.
- Werden Threads innerhalb eines Prozess-Adressraums eingesetzt, gilt diese Aussage sinngemäss auch für jeden Thread innerhalb eines Prozesskontexts.

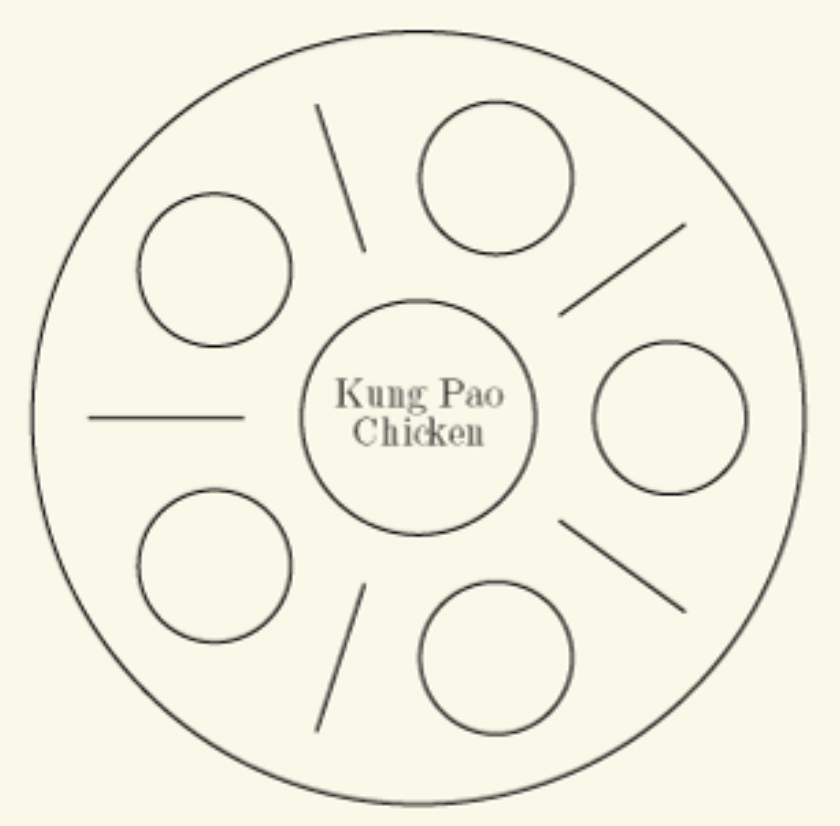
Die Rolle des Schedulers

- Der Scheduler teilt Prozessen Rechenzeit zu und sequentialisiert damit die nebenläufige Ausführung von Prozessen oder Threads auf einer oder mehreren CPUs. Bei genügender Performance des Systems geschieht dies so rasch, dass für den Benutzer der Eindruck der Parallelität entsteht.
- Neben der Ressourcenauslastung beeinflusst die Scheduling-Strategie den Grad der Quasi-Parallelität (zu häufige Wechsel → Thrashing)

Pre-emption / Re-entrance

- Prozesse können in Unix zu fast jeder Zeit unterbrochen werden:
  - freiwillig durch Systemaufruf mit Wartezustand oder Abgabe der CPU (sleep, wait);
  - ungeplant durch den Scheduler auf Basis von Priorisierung und Ressourcenverbrauch oder nach Ablauf der Zeitscheibe
  - ungeplant durch asynchronen Events (z.B. Interrupts, die durch den gerade laufenden Prozess behandelt werden müssen)
- Ein Prozess muss seinen Ausführungskontext unterbrechen, abspeichern, zum neuen Kontext wechseln, im neuen Kontext ablaufen, den alten Kontext wiederherstellen und neu starten können (Aufgabe des Kernels).
- Behandelt der Prozess im Programmcode Ausnahmen selbst (z.B. Signale), muss der Programmierer selbst für die Konsistenz von Variablen und Zuständen sorgen.

Das Problem der Synchronisierung



- 5 Personen sitzen am Tisch, zwischen den Personen liegen 5 Stäbchen
- Zum Essen benötigt jede Person 2 Stäbchen
- Problem 1: Griff in‘s Leere
  → Warten
- Problem 2: Gleichzeitiger Zugriff
  → Synchronisation
- Problem 3: Jede Person nimmt das rechte Stäbchen und wartet auf das linke
  → Deadlock
- Problem 4: alle Personen sollen in sinnvolle Frist essen
  → Starvation

Synchronisation

- Zwischen Kernel und Prozessen:
  - Signalisierung
  - Schlaf-/Wartezustand des Prozesses
  - Aufwecken & Scheduling
- Zwischen Prozessen:
  - Einseitige Synchronisation
  - Mehrseitige Synchronisation
  - Gegenseitiger Ausschluss aus kritischen Abschnitten
    - Binär
    - Typisiert (z.B. Reader / Writer)
    - Gezählt (Anzahl Prozesse im kritischen Abschnitt)

Synchronisationsmittel: Signalisierung / Benachrichtigung & Warten

- Modell 1: Prozess wartet aktiv auf das Eintreffen einer Nachricht (busy waiting) → sofortige Erkennung, aber nutzloser Verbrauch von Rechenzeit, keine nebenläufige Aktivität im Prozess möglich.
- Modell 2: Prozess wartet (schläft) auf das Eintreffen eines Wecksignals (typisiert) – der Signal-Mechanismus von Unix weckt dann (via den Kernel / Scheduler) alle auf dieses Ereignis wartenden Prozesse → kein Verbrauch von Rechenzeit, nebenläufige Aktivität im Prozess möglich.

Synchronisationsmittel: Locks/Schlossvariablen

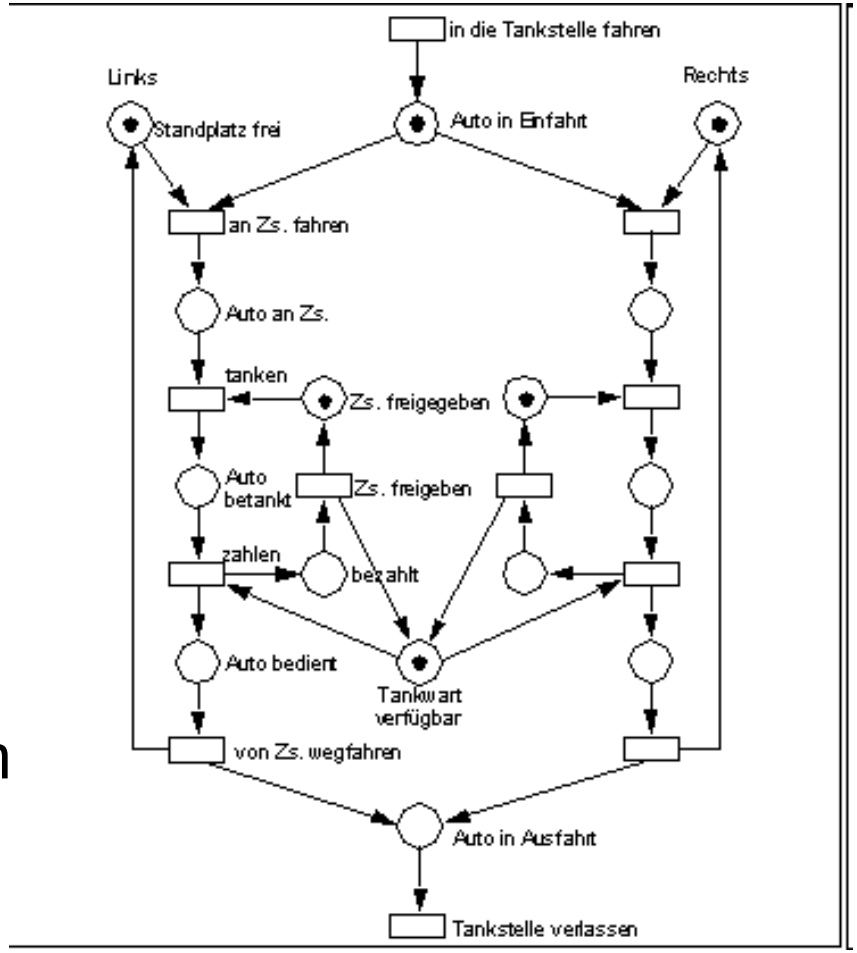
- Im Dateisystem (Lockfile, Lock-Bits im Superblock)
- Im Speicher (Lock Bits, gemeinsame Variablen)
- Modell: der Prozess prüft zyklisch auf Zustandsänderung des Bits / der Variablen und modifiziert das Bit bzw. die Variable, falls erlaubt.
- Problem: meist keine atomare Operation wegen jederzeitiger Unterbrechbarkeit, es kann daher zu Problemen kommen, wenn die Implementation nicht durch unteilbare CPU-Instruktionen („test and set“, „read and clear“, „compare and swap“ / CMPXCHG) unterstützt wird.

Synchronisationsmittel: Ereigniszähler

- Variante eines Locks, welches durch eine Zählervariable (z.B. in einem Shared Memory Segment) implementiert wird.
- Sinnvoll, wenn mehrere Prozesse zu synchronisieren sind, z.B. maximale Anzahl quasi-paralleler Leseprozesse auf einer Datenbank wegen Performance-Garantien.
- Gleiches Problem der unterbrechbaren, nicht atomaren Operation wie bei Locks, benötigt daher ähnliche Schutzmechanismen.

Synchronisationsmittel: Petri-Netze

- Modellierung von nebenläufigen Systemen und ihrer Synchronisation
  - Stellen = Kreis
  - Transitionen = Rechteck
  - Marken = Punkt
  - Schaltregeln = alle vorausgehenden Stellen enthalten mind. 1 Marke, alle nachfolgenden Stellen erhalten eine Marke



Synchronisationsmittel: Dijkstra Semaphore

- Modul/Kapsel mit geschützter Statusvariablen
- Bereitstellung von Operationen für:
  - Initialisierung
  - Eintritt/Signalisierung (P)
  - Austritt/Freigabe (V)
  - Deallokation
- Typen:
  - Binär
  - Zähler
  - Set / Array
- Implementierung: atomare CPU-Instruktion oder kurzzeitige Erhöhung des Interrupt-Levels

Semaphoren in Unix

```
P(sem)
      if (sem != 0)
         decrement sem value by one
      else
         wait until sem becomes non-zero


V (sem)
      if (queue of waiting processes not empty)
         restart first process in wait queue
      else
         increment sem value by one
```

Haviland/Salama, UNIX System Programming, Seite 191

Benutzung von Semaphoren für gegenseitigen Ausschluss

```
s: semaphore (1);

P1 : process                          P2 : process
      ...                                   ...
      P(s);                                 P(s);
      ... -- critical section               ... – critical section
      V(s);                                 V(s);
      ...                                   ...
end  process                          end  process
```

Herrtwich/Hommel, Kooperation und Konkurrenz, 1989

Benutzung von Semaphoren für einseitige Synchronisation

```
s: semaphore (0);

P1 : process                          P2 : process
      ...                                   ...
      V(s); -- signal event                 P(s); -- wait for event
      ...                                   ...
end  process                          end  process
```

Herrtwich/Hommel, Kooperation und Konkurrenz, 1989

Additive Semaphore

```
exclusion : add_semaphore (N);

     type reader = process
     ...
     P (exclusion, 1);
          ... – read within critical section
     V (exclusion, 1);
     end process;

     type writer = process
     ...
     P (exclusion, N);
          ... -- write within critical section
     V (exclusion, N);
     end process;
```

Herrtwich/Hommel, Kooperation und Konkurrenz, 1989

Semaphore Sets

- Mehrere Semaphore in einer Datenstruktur
- Manipulationen (P, V) auf einer, mehreren oder allen Semaphoren im Set erlaubt
- Garantie des Kernels, dass alle in einem Systemaufruf spezifizierten Manipulationen in einer atomaren Operation ausgeführt werden
- Implementation z.B. durch Schutz des Sets durch einen umgebenden binären Semaphor.

Synchronisationsmittel: Barrier

Die fehlerhafte Programmierung eines wechselseitigen Ausschlusses mit Semaphoren kann zu signifikanten Fehlern führen → neues Sprachkonstrukt MONITOR ohne explizite Programmierung von P und V Operationen – stattdessen Generierung durch den Compiler (Brinch Hansen & Hoare, unterstützt z.B. in Concurrent Pascal, Modula, Java).

- Modul, welches Daten und Methoden/Prozeduren enthält.
- Aufruf des Monitor Entry durch beliebig viele Prozesse; Garantie des wechselseitigen Ausschlusses
- Prozeduren/Methoden eines Monitors können auf globale Daten zugreifen, die lokalen Daten des Monitors sind aber von aussen nicht zugänglich
- Im Innern eines Monitors ist es ggf. nötig, dass eine Aktivität wartet. Eine solche Aktivität wird durch den Monitor aus dem Monitor ausgelagert. Auf diese Weise bleibt der Monitor nicht blockiert und eine andere Aktivität kann in den Monitor eintreten. Falls eine andere Aktivität den Wartenden befreit, so kann diese, sobald der Monitor frei ist, diesen wieder betreten.

Nach H. P. Oser - Betriebssysteme I, Studiengang Informatik, 2009

Synchronisationsmittel: Rendezvous

- Synchronisation von entfernten Prozeduraufrufen (remote procedure calls)
- Der Prozedur-Aufrufer wird blockiert, bis die entfernte Prozedur ausgeführt wurde
- Der Prozedur-Anbieter bleibt blockiert, bis eine seiner Prozeduren aufgerufen wird.
- Operationen:
  - Rendenzvous anbieten (Anbieter)
  - Rendezvous beantragen (Aufrufer) → Warteschlange
  - Rendezvous annehmen (Anbieter) → erster Aufrufer
  - Rendezvous ausführen & Resultat melden (Anbieter)

Deadlock Erkennung und Vermeidung/Behebung

- Deadlocks treten auf, wenn:
  - die umstrittenen Ressourcen nur exklusiv nutzbar sind,
  - die umstrittenen Ressourcen nicht entzogen werden können,
  - die Belegung von Ressourcen schon möglich ist, auch wenn auf die Zuweisung weiterer Ressourcen gewartet werden muss,
  - eine zyklische Kette von Prozessen auftritt, in der jeder Prozess mindestens eine Ressource besitzt, die der nächste Prozess in der Kette benötigt.
- Gegenmassnahmen:
  - Sicherstellen, dass immer eine der Deadlock-Bedingungen nicht erfüllt ist (Regeln für die Nutzung / erzwungene Freigabe etc).
  - Zukünftigen Ressourcenbedarf der Prozesse analysieren und Zustände erkennen/verbieten, die zu Deadlocks führen (Banker‘s Algorithm, aber ggf. schlechte Ausnutzung → Over-booking).
  - Bereits eingetretenden Deadlock erkennen und auflösen (ggf. Prozess-Abbruch).

Pause

Übung
(ca. 30 min.)

- Aufgabe(n) gemäss separatem Aufgabenblatt
- Lösungsansatz: Einzelarbeit oder Gruppen von max. 3 Personen
- Hilfsmittel: beliebig
- Besprechung möglicher Lösungen in der Klasse (es gibt meist nicht die eine «Musterlösung»)

Übungsbesprechung (ca. 15 min.)

- Stellen Sie Ihre jeweilige Lösung der Klasse vor.
- Zeigen Sie auf, warum ihre Lösung korrekt, vollständig und effizient ist.
- Diskutieren Sie ggf. Design-Entscheide, Alternativen oder abweichende Lösungsansätze.
- Gibt es Unklarheiten? Stellen Sie Fragen.

Pause

Inhalt

- Anforderungen an verteile Betriebssysteme
- Architektur und Funktionalität verteilter Betriebssysteme am Beispiel „Mach“
- Vergleich der Funktionalität von verteilten und monolithischen Betriebssystemen und deren Anwendungsgebiete
- Auswirkung von verteilten Betriebssystemen auf Datennetze, Peripheriegeräte und das System Management

Motivation

Verteilte Systeme

- Andrew Tanenbaum: „Ein verteiltes System ist ein Zusammenschluss unabhängiger Computer, der sich für den Benutzer als ein einzelnes System präsentiert.“
- Peter Löhr: „Eine Menge interagierender Prozesse (oder Prozessoren), die über keinen gemeinsamen Speicher verfügen und daher über Nachrichten miteinander kommunizieren.“
- Leslie Lamport: „A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.“

Verteilungsstrategien / -varianten

- Hardware / Software / Benutzung
- Client-Server-System: Viele Clients greifen auf einen oder mehrere Server bzw. Services zu.
- Verteiltes Dateisystem (z.B. NFS): Ein über mehrere Server verteiltes virtuelles Dateisystem steht Clients zur transparenten Benutzung zur Verfügung.
- Netzwerk-fähiges Betriebssystem (lose Kopplung): das Betriebssystem stellt system-übergreifende Funktionalität (z.B. rcp, rlogin, rsh unter Unix) zur Verfügung.
- Verteiltes Betriebssystem (enge Kopplung): Das Betriebssystem selbst ist verteilt, für Benutzer und Anwendungen ist dies nicht sichtbar.
- Verteilte Anwendung: Durch die Programmierung der Anwendung wird das verteilte System erstellt – das Programm muss in der Regel die Verteillogik kennen.

Gründe für die Verteilung

- Verteilung versus Parallelität
- Lastverteilung
- Leistungssteigerung
- Skalierbarkeit / Flexibilität
- Sicherheit / Zuverlässigkeit / Verfügbarkeit
- Preis / Leistung bzw. Kostenreduktion
- Gemeinsame Datennutzung
- Gemeinsame Nutzung teurer Peripherie
- Applikatorische Verteilung (z.B. E-Mail)
- …

Potentielle Nachteile und Risiken

- Erhöhte Komplexität
- Erschwerte Fehlersuche
- Verlängerte Abhängigkeitsketten (z.B. Netzwerk)
- Sicherheitsdispositiv wird aufwendiger
- Nicht alle Subsysteme (Hard-, Software) eignen sich für die Verteilung
- Sicherstellung der Konsistenz und Synchronisation
- …

Anforderungen an verteilte Betriebssysteme

- Gemeinsamer Kernel Code und System Calls
- Gemeinsam genutzter Hauptspeicher
- Gemeinsames Prozess-Steuersystem
- Gemeinsames Dateisystem
- Gemeinsame Interprozess-Kommunikation
- Gemeinsame Ausnahmebehandlung
- Gemeinsame Synchronisationsmechanismen
- Gemeinsames System Management
- Spezialisierte Funktionen für das Management der Verteilung
- Transparente Benutzerschnittstelle

Anforderungen an verteilte BS: Transparenz

- Ortstransparenz: Der Ort der genutzten Ressourcen / erbrachten Dienste ist für den Anwender nicht sichtbar.
- Migrationstransparenz: Ressourcen können verlagert werden, ohne dass sich ihr Name bzw. ihre Nutzung verändert.
- Replikationstransparenz: Anwender können nicht erkennen, wie viele Instanzen es gibt.
- Nebenläufigkeitstransparenz: mehrere Anwender können die Ressourcen automatisch gemeinsam und unabhängig voneinander nutzen.
- Parallelitätstransparenz: Aktivitäten können parallel bzw. Nebenläufig ausgeführt werden, ohne dass der Anwender es bemerkt.

Anforderungen an verteilte BS: Flexibilität

- Services sollen dynamisch an den Bedarf anpassbar sein, entweder durch administrative Eingriffe oder durch Eigenkonfiguration zur Laufzeit.
- Änderungen sollen keinen kompletten Neustart des verteilten Systems erfordern.

Anforderungen an verteilte BS: Zuverlässigkeit

- Erhöhte Zuverlässigkeit gegenüber Einzelsystemen trotz additiver Ausfallwahrscheinlichkeiten (Redundanz).
- End-zu-End Verfügbarkeit statt Komponentenverfügbarkeit.
- Sicherheit – gleiche Richtlinien & Umsetzung im gesamten verteilten System.
- Fehlertolerenz (versus Overhead).
- Automatisches Recovery von Komponenten.

Anforderungen an verteilte BS: Performance

- Verteilungs- und Kommunikations-Mehraufwand muss den Aufwand wert sein (Granularität, Fehlertolerenz).
- Wiederholbarkeit / Determinismus von Leistungsindikatoren.
- Abhängigkeit von nicht direkt kontrollier-baren Komponenten (z.B. LAN).
- End-zu-End Performance statt Komponen-ten-Performance.

Anforderungen an verteilte BS: Skalierbarkeit

- Angebotsseite:
  - Statische oder dynamische Zufügung / Wegnahme von Servern.
  - Verrechnung: Durchschnitt oder „peaks“?
  - Vermeiden von „Flaschenhälsen“ durch zu starke Serialisierung (Applikationskomponenten, Datenstrukturen, Algorithmen).
- Dienstnehmerseite:
  - Nicht vorhersagbare Anzahl Dienstnehmer.
  - Einhaltung von Dienstgütegarantien

Architektur und Funktionalität verteilter Betriebssysteme am Beispiel „Mach“

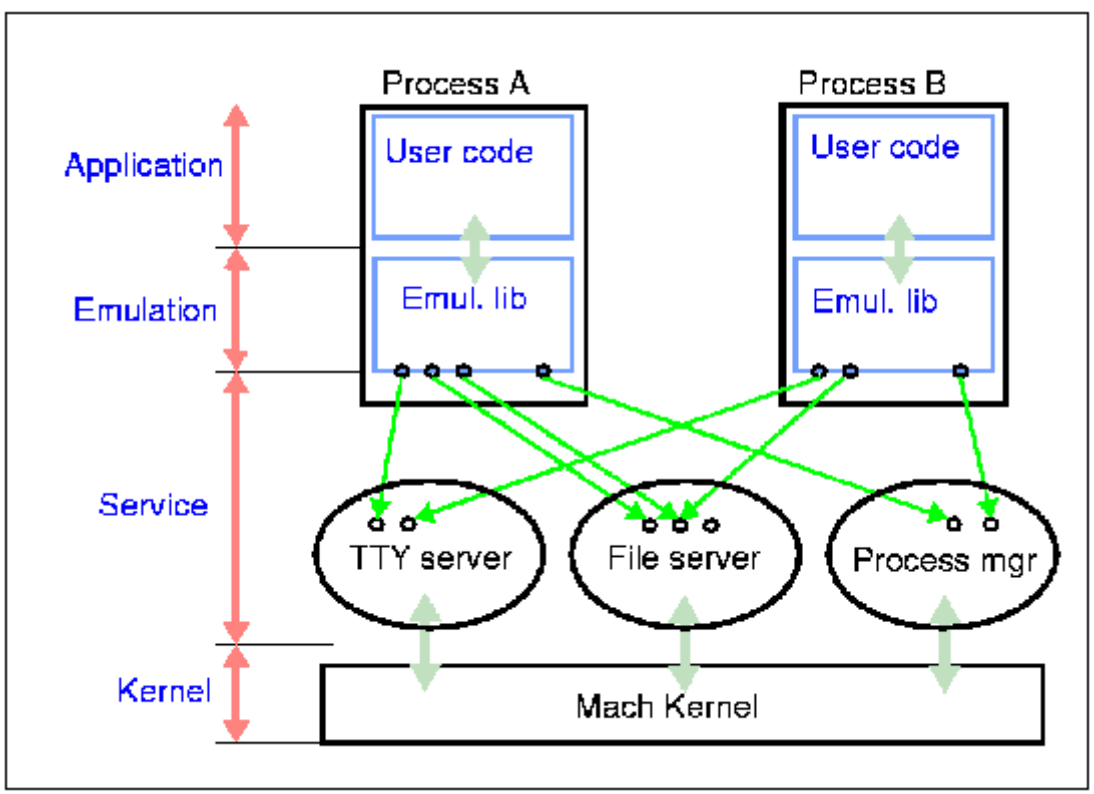
- Entwickelt an der Carnegie-Mellon Universität als Plattform für Forschung an verteilten Betriebssystemen
- Enge geistige Verwandschaft mit Unix
- Derivate: MachTen, Mac OS X, iPhone OS)
- Microkernel-Architektur (d.h. Auslagerung grosse Teile des ursprünglichen Kernel-codes in den Benutzeradressraum).

Architektur und Funktionalität verteilter Betriebssysteme am Beispiel „Mach“

- Der Kernel verwaltet 5 einfache Basiselemente:
  - Ein **Task** ist eine Umgebung zur Ausführung von Prozessen. Er besteht aus einem virtuellen Adressraum, einem oder mehreren Ports zur Kommunikation sowie mindestens einem Thread.
  - Als **Thread** wird ein einzelner Ablauf innerhalb eines Tasks bezeichnet. Ein Task kann mehrere nebenläufige Threads enthalten. Diese verwenden dann alle zum Task gehörenden Resourcen (Speicher, Ports, etc.) gemeinsam.
  - Ein **Port** ist ein einfacher Kommunikationskanal. Der Zugriff auf Ports wird vom Kernel über Rechte (port rights) kontrolliert.
  - Über Ports werden **Nachrichten** (messages) zwischen Tasks und dem Kernel ausgetauscht.
  - Über **Memory Objects** wird den Tasks Speicher zur Verfügung gestellt. Neben physischem Speicher kommen auch Dateien oder Pipes als Memory Objects in Frage. Diese werden von externen Servern (z.B. Dateisystem-Server) bereitgestellt und in den Adressraum des Tasks eingeblendet.
- Das „echte“ Betriebssystem (z.B. Unix) wird auf einer Software-Emulations-schicht bereitgestellt und läuft damit vollständig im Benutzer-Adressraum. Es können verschiedene BS nebenläufig existieren (Virtualisierung).

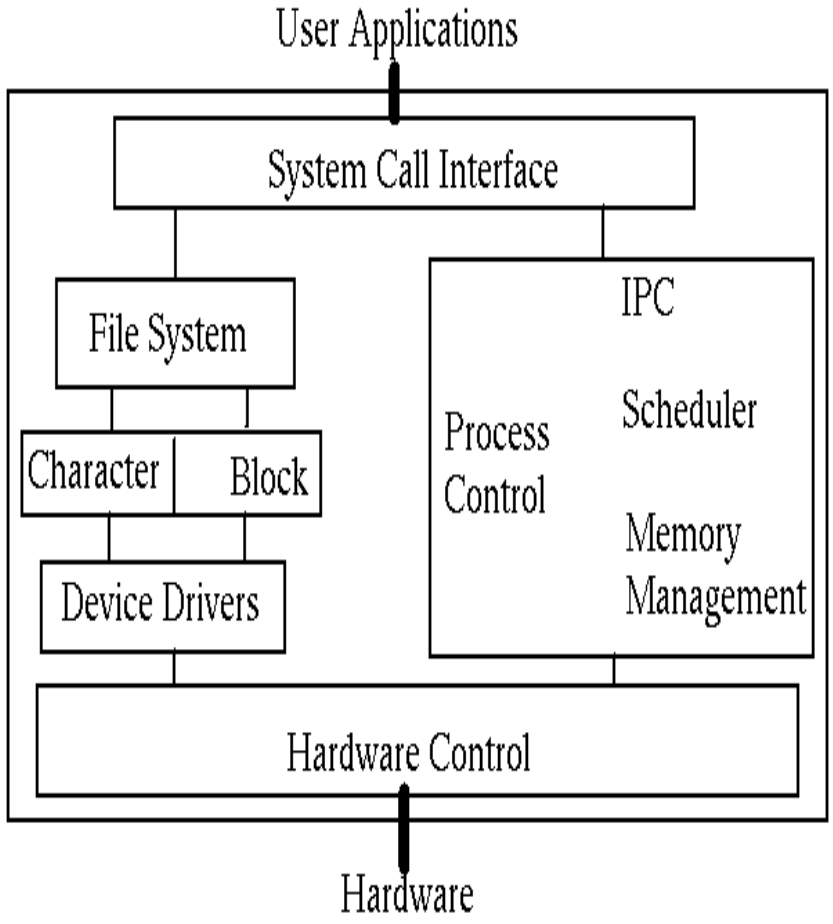
Architektur und Funktionalität verteilter Betriebssysteme am Beispiel „Mach“

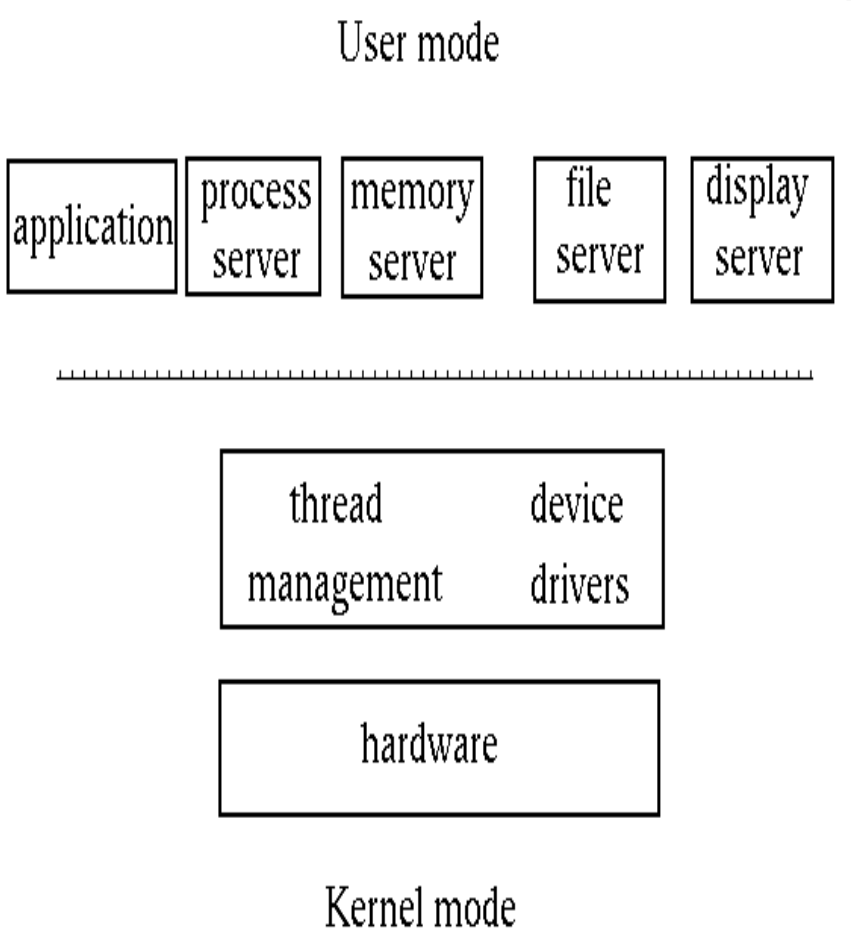
- Architektur-Übersicht



http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/projects/mach_us.html

Architekturvergleich Unix / Mach





http://jan.newmarch.name/ssw/intro/generic.html

Synchronisation in verteilten Systemen

- Zeitsynchronisation
  - Absolut
    - Zeit-Server (z..B. Network Time Protocol, NTP)
    - Eigene Referenz-Uhr (z.B. Funkempfänger)
  - Relativ
    - Median mehrerer interner oder externer Quellen
    - Neuester Zeitstempel gilt
    - Maximale Abweichung begrenzen, sonst Lokalzeit nutzen
  - Vorstellen ist einfacher als Rückstellen (Backup, „make“, …)

Synchronisation in verteilten Systemen

- Gegenseitiger Ausschluss über Systemgrenzen
  - Zentraler Algorithmus (mit Redundanz)
    - Zuverlässig, berechenbar
    - Starke Serialisierung
  - Verteilter Algorithmus
    - Zeitliche Ordnung im Gesamtsystem (setzt synchrone Uhrzeiten voraus)
    - Getimte Nachricht an alle Prozesse und Warten auf Bestätigung (setzt zuverlässige Nachrichtenübermittlung voraus)
  - Token Ring Algorithmus
    - Explizite Erlaubnis zum Zugriff in vordefinierter Reihenfolge
    - Suboptimale Ressourcennutzung / Wartezeiten

Verwaltung von gemeinsam genutztem Speicher

- Eng gekoppelt:
  - (Non-Uniform Memory Access) NUMA: Hardware-Multiprozessoren mit diversen Prozessoren am gleichen Bussystem
- Lose gekoppelt:
  - Distributed Shared Memory (DSM): Ein Seitenlade-fehler wird dadurch aufgelöst, dass eine Seite nicht von der lokalen Disk geladen wird, sondern von einer entfernten Disk
  - Schlechte Performance, lange Netz-Wartezeiten
  - Optimierung durch Anlage nur der gemeinsam benutzten Datenstrukturen auf gemeinsam genutzten Seiten, alle anderen Seiten bleiben lokal.

Verwaltung von gemeinsam genutztem Speicher

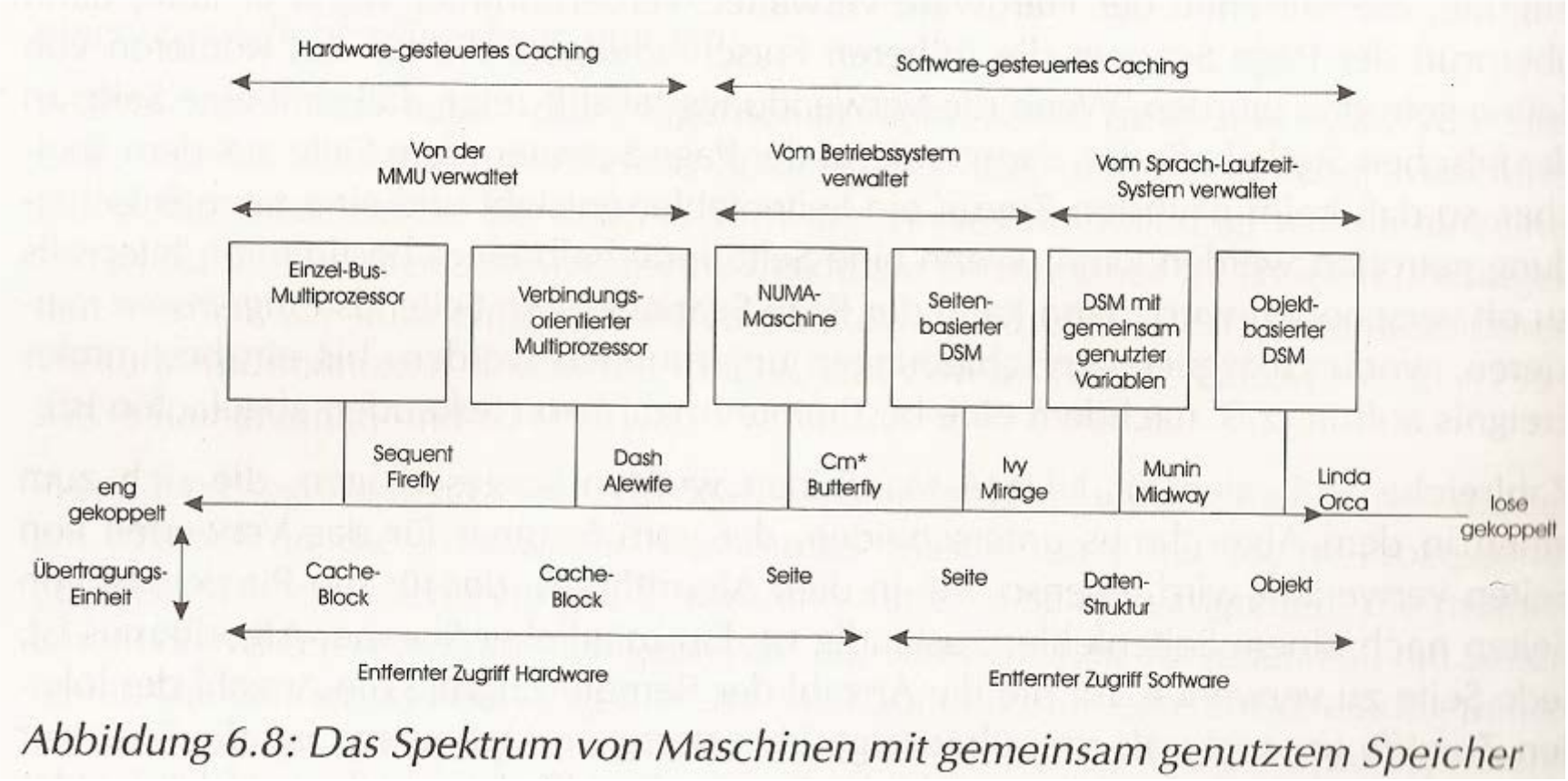


Abbildung 6.8: Das Spektrum von Maschinen mit gemeinsam genutztem Speicher

Andrew Tanenbaum: Verteilte Betriebssysteme, Prentice Hall

Vergleich der Funktionalität von verteilten und monolithischen BS

- Ein monolithisches Betriebssystem präferiert Effizienz vor Robustheit. Obwohl das Design die Trennung funktionaler Blöcke vorsieht, werden Privilegien nicht auf Teile des Betriebssystems beschränkt (z.B. „root“ Ausführungsrechte im Datei- und Prozesssystem). Die verwendeten Kommunikationsmechanismen (Signale, Pipes, Share Memory, Sockets usw.) sind überall identisch.

- Eine Schichtenarchitektur des Betriebssystems fördert die Robustheit durch die Anordnung von Funktionalität in gegeneinander abgegrenzte Schichten mit unterschiedlichen Privilegien. Der höchstprivilegierte Block behandelt Interrupt-Behandlung und Kontextwechsel, die darüberliegenden Schichten beinhalten Gerätetreiber, Speicherverwaltung, das Dateisystem und die Benutzerschnittstelle. Am oberen Ende folgt schliesslich die am wenigsten priorisierte Applikationsschicht.

- Eine Microkernel-Architektur präferiert Robustheit vor Effizienz. Die den einzelnen Teilen des Betriebssystems zugeordneten Privilegien sind so restriktiv wie möglich und die Kommunikation zwischen den Blöcken basiert auf spezialisierten Kommunikationsmechanismen, die auf die jeweiligen Zugriffsrechte begrenzt sind (nur wenige Teile des Betriebssystems benötigen mehr Rechte als die Applikationen). Somit kann das Betriebssystem oberhalb des Microkernels auf wenige Service begrenzt werden, die die Basisfunktionen des Betriebssystems realisieren.

Auswirkung von verteilten Betriebssystemen

- Datennetze
  - Hochpriorisierter Datenverkehr im LAN
  - Hohe Zuverlässigkeit, niedrige Latenz
  - Schutz vor Mitlesen / Modifikation nötig
- Peripheriegeräte
  - Geteilte Nutzung über Systemgrenzen
  - Verwaltung / Scheduling nötig
- System Management
  - Beobachtung einzelner Systeme genügt nicht
  - Dynamische Konfigurationsänderungen

Zusammenfassung der Lektion 11 und Hausaufgabe

- Die Problematik der Nebenläufigkeit von Prozessen und des Bedarfs für Prozess-Synchronisation.
- Deadlock-Szenarien erkennen und Strategien zu deren Vermeidung bzw. Behebung beschreiben können.
- Zusammenhang zwischen Nebenläufigkeit und Prozess-Synchronisation einerseits, und Scheduling bzw. Ressourcenverwaltung andererseits.
- Unterschiede zwischen monolithischen und verteilten Betriebssystemen sowie jeweilige Vor- und Nachteile / Einsatzgebiete.
- Parameter für die optimale Verteilung von Betriebs-systemleistungen und deren Beurteilung und Anwendung.
- Auswirkung der Verteilung von Betriebssystemleistungen auf die umgebende ICT-Infrastruktur beurteilen.
- Hausaufgabe:
  - Repetieren Sie den Stoff dieser Lektion.
  - Studieren Sie das Dokument „11-VertSys2000_1.pdf“
  - Studieren Sie das Material unter den Link „http://de.wikipedia.org/wiki/Lock“,„http://de.wikipedia.org/wiki/Nebenläufigkeit“ sowie die Folge-Links unter der Sektion „Siehe auch“.