2. Betriebssystemstrukturen

https://www.freecodecamp.org/news/building-and-installing-the-latest-linux-kernel-from-source-6d8df5345980/

Notizen

Betriebssystemstrukturen

- Betriebsmodi
- Monolithische Systeme
- Client / Server-Strukturen (Mikrokerne)
- Virtuelle Maschinen
- Zusammenfassung

Notizen

Benutzungs- und Kernmodus

Systemsoftware
bietet „Dienste" durch:

- Bibliotheken / Middleware
- Kern-Dienste (Systemcalls)

Rechnerarchitektur besitzt
(mind.) 2 Betriebsmodi:

- Nicht-privilegiert:
  Benutzungsmodus
- Privilegiert:
  Kernmodus

Übergang durch Systemaufruf
(= „TRAP"-Befehl)

Systemaufruf-Schnittstelle
(= „system call interface")

Notizen

Durchführung eines Kernaufrufs

Benutzungsprogramme und

Betriebssystem befinden sich im

Arbeitsspeicher

Anwendungen laufen im
Benutzungsmodus

Das Betriebssystem läuft im
Kernmodus

Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.

BS Code bestimmt die Nummer des angeforderten Dienstes.

BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.

Kontrolle wird an das Anwendungsprogramm zurückgegeben.

Wichtig: Kern selbst ist passiv (Menge von Datenstrukturen und Prozeduren)

Notizen

Systemaufruf-Beispiel

Hochschule RheinMain

„Stub“ Code der Funktion `open()` in der Linux C-Bibliothek [1]

```
int open(char *name, int flags, mode_t mode)
```

```
...
int fd;
fd = open("myfile",
          O_RDWR|O_CREAT,
          0777);
if(fd < 0)
    perror("open()");
...
```

```
__libc_open:
   push   %ebx
   mov    0x10(%esp),%edx
   mov    0xc(%esp),%ecx
   mov    0x8(%esp),%ebx
   mov    $0x5,%eax
   int    $0x80
   pop    %ebx
   cmp    $0xfffff001,%eax
   jae    __syscall_error
   ret
```

Argumente (vom Stack) in Register

Funktionsnummer: 5 = open()

TRAP-Befehl

[1] Hinweis: UNIX Manual
- Systemaufrufe in Manual Section 2 (z.B.:`open(2)`, `close(2)`, `read(2)`, `write(2)`, ...)
- C-Bibliotheksfunktionen in Manual Section 3 (z.B.:`printf(3)`, `scanf(3)`, `malloc(3)`, ...)

Notizen

Betriebsmodi

Hochschule RheinMain

Aus Hardware-Sicht: Betriebsarten (Modi) des Prozessors

Bei den meisten Architekturen: zwei Stufen
(Vier bei x86, die meisten BS nutzen aber nur zwei davon)

- privilegiert
- nicht-privilegiert

Privilegierter Modus erlaubt Zugriff / Manipulation der „Maschinenkonfiguration“

- Sperren von Unterbrechungen
- Zugriff auf Speicherverwaltungs-Hardware
- Privilegierte Maschinenbefehle

Übergang in höhere Stufe durch Ausnahmebedingungen („Exceptions“)

- Unterbrechungen („Interrupts“)
- Explizite TRAP-Befehle
- Schutzverletzungen („Faults“)

Rückkehr zu niedrigerer Stufe durch spezielle Maschinenbefehle

Notizen

Betriebsmodi (3)

Hochschule RheinMain

Aus Betriebssystemsicht

Im Benutzungsmodus (= nicht-privilegierten Modus):

- Beschränkter Zugriff auf Betriebsmittel
- Zugriff auf nicht-zugeteilte BM[2] löst Fault aus
- Unerlaubte Operation[3] löst Fault aus
- Systemcall (= expliziter TRAP-Befehl) löst TRAP aus

Im Kernmodus (= privilegierten Modus):

- Uneingeschränkter Zugriff auf alle Betriebsmittel
  (→ Ist auch notwendig zu deren Verwaltung)
- Unerlaubte Operation oder Schutzverletzung im Kernmodus ⇒ ☠!

⇒BS-Kern-Code <u>muss</u> fehlerfrei sein!

[2] „Schutzverletzung“
[3] z.B. Division durch Null, illegaler Befehl, alignment-Fehler...

Notizen

Monolithische Systeme

Hochschule RheinMain

Vorwiegende Struktur aller kommerzieller Betriebssysteme (z.B. UNIX, Windows)

Durch Binder zu einem „Klumpen“ zusammengebunden (bis auf dyn. ladbare Kernmodule)

Notizen

Einfaches Strukturmodell

Hochschule RheinMain

Innere Struktur eines monolithischen BS:

Da der Betriebssystemkern passiv ist
und der Code aus einer Menge von Prozeduren besteht, heißt ein solches Betriebssystem auch prozedurorientiert.

Notizen

Beispiel: UNIX

Hochschule RheinMain

Blockdiagram des Systemkerns[4]

[4] aus [Bach]: The Design of the UNIX Operating System

Notizen

Notizen

Beispiel: UNIX Codeumfang

Hochschule RheinMain

Der UNIX-Betriebssystemkern

monolithisch, aber portierbar

Beispiel: 4.3BSD UNIX Kern (1987)

- Lines of Code 116.470 (nicht mehr!)
- C-Anteil 97,1%
- maschinenunabhängig 41,5%
- maschinenabhängig 58,5%
  davon Gerätetreiber 35,5%
  Netzwerktreiber 14,8%

Neben dem Betriebssystemkern wird ein Großteil der UNIX-Systemfunktionalität durch sogenannte Dämon-Prozesse erbracht.

Beispiel: Linux Kern SLOC (ohne Leerzeilen, ohne Kommentarzeilen):

Linux 1.0.0	(1994)	176.250
Linux 2.2.0	(1999)	1.800.847
Linux 2.6.0	(2003)	5.929.913
Linux 3.2	(2012)	14.998.651

Windows Server 2003 (Gesamtsystem) ca. 50 Mio Zeilen

Notizen

Vergleich zu sonstigen Codegrößen[6]

Project	No. of Files	eLOC[5]
Linux Kern 2.6.17	15.995	4.142.481
Firefox 1.5.0.2	10.970	2.172.520
MySQL 5.0.25	1973	894.768
PHP 5.1.6	1316	479.892
Apache Http 2.0.x	275	89.967

[5] The effective lines of code (eLOC) are measured using the following method:
1. Get the number of lines of code
2. Subtract whitespace lines
3. Subtract comment lines
4. Subtract the lines that contains only block constructs

[6] `http://msquaredtechnologies.com/m2rsm/rsm_software_project_metrics.htm`

Notizen

Client / Server-Strukturen (Mikrokerne)

Problem monolithischer Systeme: Kern wird immer **umfangreicher und komplexer**, damit zwangsläufig auch **fehlerträchtiger**.

Aller Code, der im privilegierten Modus läuft, hat Zugriff auf alle Betriebsmittel und zählt damit immer zur **„Trusted Code Base“**.

Nicht alle Anwendungen benötigen wirklich alle Dienste, die ein Kern anbietet

Art und Anzahl der Dienste werden aber durch den Kern fest vorgegeben

Mikrokern-Ansatz: Alle Dienste, deren Funktion technisch auch ohne privilegierte Operationen realisiert werden kann, werden aus dem Kern ausgelagert

Notizen

Mikrokern-Ansatz

Hochschule RheinMain

Dienste wie Dateisystem, Netzwerkprotokolle, Speicherverwaltung, Prozesssteuerung, sogar Gerätetreiber müssen nicht zwangsläufig im Kern angesiedelt sein.

Auslagerung großer Teile der Funktionalität eines BS-Kerns in Programme auf Anwendungsebene.

„Server"-Prozesse, die wie Anwendungsprogramme ohne besondere Privilegien arbeiten, bieten diese Dienste (*Services*) an.

Als „gewöhnliche" Anwendungen haben Server nur jeweils Zugriff auf die ihnen zugewiesenen Betriebsmittel.

Übrig bleibt ein **minimaler** Kern, als *Mikrokern* bezeichnet

Dieser bietet nur noch Dienste zur Kommunikation zwischen Klienten (Anwendungen) und Servern untereinander an.

Notizen

Mikrokern

Hochschule RheinMain

Client / Server Architektur

Dienste werden durch Nachrichten per **Interprozesskommunikation** (*inter process communication* – IPC) von Servern angefordert. (*send & receive*)

Server liefern Dienste ebenfalls mit IPC-Nachrichten. (*reply & wait*)

Weitere Konzepte im μKern: Adressräume und Threads

Notizen

Client / Server-Struktur (2)

Hochschule RheinMain

Vorteile:

Isolation der „Systemteile“ gegeneinander
z.B.: Ausfall eines Servers betrifft nur dessen Klienten
→ Klienten können ihre „Trusted Code Base“ feingranular auswählen
Erweiterbarkeit, Anpassungsfähigkeit und flexible Konfigurierbarkeit
z.B.: Mehrere unterschiedliche, sogar konkurrierende Dienste können gleichzeitig betrieben werden.
Nachrichtenbasiert: Prinzipielle Möglichkeit, Klienten und Server transparent auf verschiedenen Knoten eines Verteilten System zu betreiben.

Ein Betriebssystem, das auf einem über Nachrichten realisierten Beauftragungsprinzip beruht, heißt auch nachrichtenorientiert. Nachrichtenorientiertheit und Prozedurorientiertheit sind funktional gleichwertig, nachrichtenorientierte Systeme leiden aber häufig unter Ineffizienz.

Notizen

Policy and Mechanism

Hochschule RheinMain

Kriterium für auszulagernde Dienste: **Trennung von Strategie und Mechanismus** („*separation of policy and mechanism*“)

Beispiel: Speicherverwaltung:

- **Strategie** (*policy*): Zuteilung von Speichersegmenten an Prozesse
- **Mechanismus** (*mechanism*): Konfiguration der Hardware-Speicherverwaltungseinheit (*Memory Management Unit* – MMU)

μKern-Implementierung:

- Der IPC-Dienst überträgt neben Daten auch Rechte (→ z.B. Zugriffsrechte auf Speicherbereiche)
- Wird solch ein Zugriffsrecht übertragen, so konfiguriert der μKern bei der Übertragung die Hardware entsprechend.

⇒ Strategie im Server, Mechanismus im μKern.

Ein μKern *sollte* klein und wenig komplex sein

Das ist jedoch kein hinreichendes Kriterium
(Mikrokern $\neq$ kleiner Kern!)

Entscheidend ist die (möglichst weitgehende) Policy-Freiheit

Notizen

Single Server

Ansatz: Bestehendes, monolithisches BS in Server umwandeln
→ („*OS personality*“)
Anwendungen finden die gleichen Dienste vor, können unverändert bestehen bleiben
Vorteil: Mehrere Betriebssysteme in einem Rechner
Nachteil: Große Trusted Code Base (...und Performance-Verlust)

Notizen

Beispiele

Forschung

Minix / Minix3 (VU Amsterdam)

Singularity (Microsoft Research)

EROS/CoyotOS (Johns Hopkins University)

L4 Microkernel Familie

- Ursprünglich: Jochen Liedtke, GMD
- Weiterentwicklungen:
  - Uni Karlsruhe: L4Ka, Pistachio
  - TU Dresden: Fiasco, Nova Hypervisor
  - UNSW Sydney: seL4

Kommerziell

QNX Neutrino (Blackberry QNX)

Chorus OS (Chorus Systems)

PikeOS (SYSGO AG)

Notizen

Virtuelle Maschinen

Hochschule RheinMain

Trennen der Funktionen „Mehrprogrammbetrieb“ und „erweiterte Maschine“

Virtualisierung durch *„Virtual Machine Monitor“* (auch: *„Hypervisor“*)

virtuelle Maschinen als mehr oder weniger identische Kopien der unterliegenden Hardware

In jeder virtuellen Maschine: übliches Betriebssystem.

Notizen

Beispiel: VM/370

Hochschule RheinMain

1970: Das offizielle IBM-Produkt für Timesharing-Betrieb der /360, TSS/360, kam zu spät, war zu groß und zu langsam.

In der Zwischenzeit: IBM Scientific Center Cambridge, Mass. Eigenentwicklung, wurde als Produkt (ursprünglich CP/CMS) akzeptiert, erlangte als VM/370 weite Verbreitung.

Unterste Ebene: virtuelle Maschinen als identische Kopien der unterliegenden Hardware mit Nachbildung von Anwendungs- und Supervisor-Modi, I/O, Unterbrechungen,

Simulation mehrerer /370 Rechner.

Effizient durch gegebene „Virtualisierbarkeit“ der /370 Architektur.

Notizen

Beispiel: VM/370 (2)

Hochschule RheinMain

Betriebssysteme in virtuellen Maschinen: z.B. ein Stapelverarbeitungssystem (OS/360) und eine Menge von Einbenutzer-Dialogsystemen (CMS, Conversational Monitor System) gleichzeitig möglich.

Heute: z/VM: erlaubt z.B. 100 unabhängige Linux-Systeme auf einem IBM Mainframe.

Notizen

Virtualisierbarkeit

Hochschule RheinMain

Anforderung: **Identisches Verhalten der VM**.

→ Ein ausgeführtes Programm kann nicht feststellen, ob es von einer VM oder einer realen Maschine ausgeführt wird.

Möglichkeiten dazu;

- **Emulation**: Komplette Nachbilung der Hardware in Software
  → Ineffizient!
  Beispiele: Bochs (x86), JVM
- **Virtualisierung**: Nur ein geringer Teil der Befehle muss emuliert werden, die meisten Befehle werden von der realen Hardware ausgeführt.
  → Annähernd keine Effizienzeinbußen
  → Voraussetzung: Architektur muss „virtualisierbar“ sein.
  (Der x86 war das zunächst nicht!)
  Beispiele: VM/370 (s.o.), Qemu, VirtualBox, VMware
- **Paravirtualisierung**: Falls nicht virtualisierbar: Privilegierte Befehle des Gast-BS durch „Hypercalls“ (= Aufrufe in den Hypervisor) ersetzen.
  → Effizienz wie Virtualisierung (u.U. sogar noch besser)
  - Nachteil: Quellcode des Gast-BS muss angepasst werden.
  Beispiele: Xen, KVM, OpenVZ, Hyper-V

Notizen

Beispiel: VMware Workstation, Virtualbox

Erlaubt beliebige Betriebssysteme für x86-Architektur auf Linux oder Windows

Jedes Gastbetriebssystem kann abstürzen, ohne den Rest zu beeinflussen

Notizen

Beispiel: VMware Server, Xen

Xen: Paravirtualisierung: Gastsysteme müssen angepasst werden (Quellcode Voraussetzung)

VMware Server: klassischer VM Monitor (eigentlich: „JIT-Paravirtualisierung“)

Notizen

Zusammenfassung

Hochschule RheinMain

Grundverständnis einer Betriebssystemschnittstelle

Strukturierungsprinzipien von Betriebssystemen:

- Monolithische Struktur
- Client/Server-Struktur (Mikrokernel)
- Virtuelle Maschinen

Notizen

Notizen

3. Prozesse und Threads

https://www.animierte-gifs.net/data/media/1798/animiertes-richter-bild-0001.gif

Notizen

Prozesse und Threads

Prozessmodell
Implementierung von Prozessen
Threads
Zusammenfassung

Notizen

Prozessmodell

Hochschule RheinMain

Definition: Prozess (engl. *process*)

Ein **Prozess** ist ein sich in Ausführung befindliches Programm (einschließlich seiner aktuellen Werte des Programmzählers, der Register, Speichervariablen, Stack).

Ein Prozess besitzt einen privaten Adressraum

- Menge von (virtuellen) Adressen, von Prozess zugreifbar
- Programm und Daten in Adressraum sichtbar.

Verhältnis Prozess - Prozessor

- Prozess besitzt konzeptionell einen eigenen virtuellen Prozessor
- Reale(r) Prozessor(en) wird / werden zwischen den virtuellen Prozessoren umgeschaltet (Mehrprogrammbetrieb).
- Die Umschaltungseinheit heißt **Scheduler** oder **Dispatcher**, der **Scheduling-Algorithmus** legt die Regeln der Umschaltung fest.
- Der Umschaltvorgang heißt **Prozesswechsel** oder **Kontextwechsel** (*Context Switch*).
- Multicore-Prozessoren enthalten mehrere Prozessoren auf einem Chip

Notizen

Zur Verdeutlichung

Hochschule RheinMain

(a) Mehrprogrammbetrieb mit 4 Programmen

(b) Konzeptionelles Modell: 4 unabhängige, sequenzielle Prozesse

(c) Gantt-Diagramm: (1 Prozessor)

zu jedem Zeitpunkt nur 1 Programm aktiv

Notizen

Programm ≠ Prozess!

Hochschule RheinMain

Programm:

feststehende Beschreibung eines sequenziellen Algorithmus (→ „Rezept“)

Prozess:

Ausführung der Beschreibung (→ „Vorgang“)

Ein und dasselbe Programm kann mehrmals (auch gleichzeitig!) innerhalb verschiedener Prozesse ausgeführt werden.

Notizen

Beachte

Hochschule RheinMain

Ein Prozess besitzt einen privaten **Adressraum**,

hat einen **Prozesszustand**, beschrieben durch

- Programm
- Daten
- Momentane Arbeitsposition,

kann i.d.R. jederzeit unterbrochen ...

→ Prozesszustand speichern

...und später fortgeführt werden

→ Prozesszustand wiederherstellen

Die Ausführungsgeschwindigkeit eines Prozesses ist daher **nicht gleichmäßig** und **nicht reproduzierbar**.

→ Bei der Programmierung sind keine a-priori-Annahmen über den zeitlichen Verlauf zulässig.

→ Bei zeitkritischen Anforderungen, z.B. Echtzeit-System, sind besondere Vorkehrungen im Scheduling-Algorithmus notwendig.

Notizen

Prozesserzeugung

Hochschule RheinMain

Einfachster Fall: Feste Menge von Prozessen wird beim Systemstart erzeugt

Einfache, meist *eingebettete* Systeme

Beispiele

- Autoelektronik (Motorsteuerung, etc.):
  Betriebssystem „OSEK/VDX“[1]: bietet **keine** Funktionen zur dynamischen Prozesserzeugung, stattdessen offline-Konfiguration in spezieller Sprache („OIL“)
- Videorekorder , settop-Box, Handy (einfache), Weltraumsonde ...

Vorteil: geringer Verwaltungsaufwand, deterministisches Zeitverhalten

Nachteil: unflexibel

[1] heute: „AUTOSAR Classic“

Notizen

Prozesserzeugung (2)

Hochschule RheinMain

Bei komplexeren Systemen werden neue Prozesse im Laufe der Zeit dynamisch erzeugt, z.B.

Beim **Systemstart**

- z.B. UNIX-„Daemons“: Hintergrundprozesse zum Annehmen von E-Mail, Druckjobs, Web-Anfragen, ...

durch **andere Prozesse** per Systemfunktion
(z.B. „Hilfsprozess“ erzeugen)

durch **Benutzende** veranlasst

- z.B. Programmstart: „Prozesserzeugung per Doppelklick“ oder zur Abarbeitung von **Batch-Jobs** (auf Großrechnern)

Impliziert die Bereitstellung geeigneter Systemaufrufe durch das Betriebssystem

Notizen

Prozessende

Hochschule RheinMain

Freiwilliges Prozessende (Prozess beendet *sich selbst*)

Normale Beendigung

- Prozess ist „normal“ durchgelaufen
- z.B.: Aufruf von `exit(EXIT_SUCCESS)`

Beendigung aufgrund eines Fehlers

- z.B. angegebene Datei kann nicht geöffnet werden, Programm sieht Ausgabe einer Fehlermeldung und Prozessende vor
- `exit(EXIT_FAILURE)`

Unfreiwilliges Prozessende (Prozess *wird* beendet)

- Beendigung aufgrund eines schweren Fehlers, z.B.
  - Zugriff auf unzulässige Speicheradresse
  - Division durch Null

Beendigung durch anderen Prozess

- ein anderer Prozess hat mit Hilfe einer Systemfunktion das Betriebssystem überzeugt, den Prozess abzubrechen.

Notizen

Prozesshierarchie

Hochschule RheinMain

Manche Systeme merken sich Zusammenhang zwischen erzeugendem Prozess (Elternprozess) und von diesem erzeugtem Prozess (Kindprozess)

Prozessfamilie: Prozess und alle seine Nachkommen

Prozesshierarchie: Baum-strukturierte Prozess-Menge (z.B. UNIX)

Gegenbeispiel Windows:

- keine Hierarchie,
- alle Prozesse sind gleichwertig,
- erzeugender Prozess erhält Verweis („Handle“) auf erzeugten Prozess,
- dieses Handle kann er jedoch beliebig weitergeben ($\rightarrow$ nicht notwendig Baumstruktur)

Notizen

Beispiel: UNIX Systemstart

Hochschule RheinMain

Beim UNIX-Systemstart wird der Prozess `init` (Prozess Nr. 1) erzeugt (Parent aller nachfolg. Prozesse)

`init` führt eine Reihe von Programmen (i.d.R. Shell-Scripts) aus dem Verzeichnis `/etc/init.d` aus.

Dabei werden unter anderem auch Anmelde-Prozesse gestartet

Melden sich Benutzende an, wird jeweils ein Shell-Prozess erzeugt, der wiederum bei Kommandoeingaben entsprechende Unterprozesse erzeugt usw.

UNIX-Kommandos zur Ausgabe der Prozessliste:

- `ps` Standard-Kommando
- `pstree` baum-formatierte Ausgabe (nicht überall verfügbar)

Notizen

Prozesszustände

Hochschule RheinMain

Prozesse, obwohl unabhängige Einheiten, können aufgrund des Algorithmus logisch voneinander abhängig sein.

Beispiel (UNIX Shell):`cat datei1 datei2 datei3 | grep hugo`

In Abhängigkeit von den relativen Ausführungsgeschwindigkeite kann ein Prozess warten müssen, bis eine Eingabe vorliegt.

Allgemeiner sagt man: Er **blockiert** und wartet auf ein (für ihn externes) **Ereignis**.

Prozessor wird dann unmittelbar einem anderen Prozess zugeordnet. Entzug des Prozessors (**Suspendierung**) in diesem Fall problembegründet.

Auch möglich: Scheduler entscheidet auf Prozesswechsel, obwohl der erste Prozess weiter ausgeführt werden könnte (**Preemption**).

Notizen

Prozesszustände (2)

Hochschule RheinMain

Damit sinnvolle Prozesszustände:

rechnend (oder **aktiv**):dem Prozess ist ein Prozessor zugeordnet, der das Programm vorantreibt.

rechenwillig (oder **bereit**):Prozess ist ausführbar, aber alle verfügbaren Prozessoren sind anderen Prozessen zugeordnet.

blockiert (oder **schlafend**): Prozess wartet auf Ereignis. Er kann solange nicht ausgeführt werden, bis das Ereignis eintritt.

Gelegentlich noch folgende Zustände:

initiiert: in Vorbereitung (Anfangszustand).

terminiert: Prozess ist beendet (Endzustand).

Notizen

Zustandsübergangsdiagramm

Hochschule RheinMain

Zustandsübergänge:

1	rechnend → blockiert:	Versetzung in den Wartezustand (Suspension, warten auf Ereignis)
2a	rechnend → rechenwillig:	Scheduler entzieht Prozessor. (Preemption)
2b	rechnend → rechenwillig:	Prozess gibt Prozessor freiwillig ab. (Yield)
3	rechenwillig → rechnend:	Scheduler teilt Prozessor zu.
4	blockiert → rechenwillig:	Ereignis tritt ein.

Notizen

Zusammenfassung

Hochschule RheinMain

Das Prozessmodell vereinfacht die Beschreibung der Aktivität des Rechensystems.

Die ineinander verwobene Aktivität des Systems wird durch eine Menge von sequentiellen Prozessen beschrieben.

Die unterste Schicht eines Betriebssystems behandelt die Unterbrechungen und ist für das Scheduling verantwortlich. Der Rest des Systems besteht aus sequentiellen Prozessen.

Notizen

Anmerkungen

Hochschule RheinMain

Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig → Kap. 5, 6

Programmierung von Anwendungen aus mehreren nebenläufigen Prozessen heißt Concurrent Programming.

Das klassische Prozessmodell wird verfeinert durch Einführung sogenannter **Leichtgewichtsprozesse** (light weight processes, oder **Threads** = Fäden), die mehrere Aktivitätsträger in einem einzigen Adressraum darstellen → 3.3.

Notizen

Implementierung von Prozessen

Hochschule RheinMain

Datenstrukturen im BS-Kern zur Prozessverwaltung:

PCB dient zum Speichern des gesamten Zustandes eines Prozesses

Notizen

Prozesskontrollblock

Hochschule RheinMain

Typische Inhalte eines Prozesskontrollblocks:

Prozessverwaltung
Register
Programmzähler
Programmstatuswort
Stack-Zeiger
Prozesszustand
Prozessnummer
Elternprozessnummer
Prozesserzeugungszeitpunkt
Terminierungsstatus
verbrauchte Prozessorzeit
Prozessorzeit der Kinder
Alarm-Zeitpunkt
Signalstatus
Signalmaske
unbearbeitete Signale
Zeiger auf Nachrichten
verschiedene Flags

Speicherverwaltung
Zeiger auf Textsegment
Zeiger auf Datensegment
Zeiger auf BSS-Segment
Prozessgruppe
reale UID
effektive UID
reale GID
effektive GID
verschiedene Flags

Dateisystem
Wurzelverzeichnis
aktuelles Verzeichnis
UMASK-Maske
offene Dateideskriptoren
effektive UID
effektive GID
Aufrufparameter
verschiedene Flags

zusätzlich: Zeiger zur Verkettung des PCB in (verschiedenen) Warteschlangen

Notizen

Multitasking

Notizen

Warteschlangenstruktur

Einfache Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder Ready Queue):

Bei gleicher Priorität: Einreihen nach „first in / first out“
Nachteil: Laufzeit abhängig von Prozessanzahl

Notizen

Warteschlangenstruktur

Hochschule RheinMain

Typische Struktur der Liste der rechenwilligen Prozesse (Bereit-Liste oder Ready Queue):

Eine Warteschlange pro Prioritätsstufe

Schnelles Einreihen in konstanter Laufzeit („O(1)-Scheduler“)

(Nachteil: Begrenzte Anzahl möglicher Prioritäten)

Notizen

Wer (oder was) aktiviert den Scheduler?

Hochschule RheinMain

Wenn der Scheduler die Kontrolle an einen ausgewählten Prozess abgibt - wie bekommt er sie dann wieder zurück?

Ein Ansatz: Jedes Programm führt „oft genug“ einen Systemaufruf aus, das Betriebssystem ruft bei jedem Systemaufruf den Scheduler auf.

- **„kooperatives Multitasking“**
- z.B. in früheren Windows- und MacOS-Versionen
- Nachteil: Ein Prozess kann nicht gezwungen werden, die Kontrolle abzugeben; Problem bei „bösen“ Programmen.

Alternative: Preemptives Multitasking

- benötigt Hardware-Unterstützung
- Bei Eintreten bestimmter Ereignisse (Ein-/Ausgabe, Ablauf eines Timers, ...) wird der gerade laufende Prozess „von außen“ unterbrochen und Code zur Unterbrechungs-Verarbeitung aufgerufen
- hierbei kann Aufruf des Schedulers vorgesehen werden

Notizen

Unterbrechungslogik

Notizen

Unterbrechungsbehandlung

Interrupt-Handler: Gerätespezifische Routinen zur Behandlung von Interrupts

Zuordnung von Interrupt(-Nummer) zu Interrupt-Handler über *Interrupt-Vektor-Tabelle* (IVT)

Notizen

Unterbrechungsbehandlung (2)

Hochschule RheinMain

Typische Ausführungsschritte:

- Programmzähler (u.a.) wird durch Hardware auf dem Stack abgelegt.
- Hardware lädt den neuen Programmzählerinhalt aus dem Unterbrechungsvektor.
- Eine Assembler-Routine rettet die Registerinhalte.
- Eine Assembler-Routine bereitet den neuen Stack vor.
- Eine C-Prozedur markiert den unterbrochenen Prozess als rechenwillig.
- Der Scheduler bestimmt den Prozess, der als nächster ausgeführt werden soll.
- Die C-Prozedur gibt die Kontrolle an die Assembler-Routine zurück.
- Die Assembler-Routine startet den ausgewählten Prozess.

Notizen

Interrupts aus Sicht des Prozesses

Hochschule RheinMain

Laufender Prozess (A) wird unterbrochen

Interrupt Service Routine (ISR) wird ausgeführt

Weiterführen, entweder Prozess (A), oder zuvor blockierter Prozess (B) (der durch ISR „rechenwillig“ wurde)

Relevante Zeitspannen:

IRT *Interrupt Response Time*
PDLT *Process Dispatch Latency Time*
SWT *Process Switch Time*

Notizen

Beispiel UNIX

Hochschule RheinMain

Systemaufrufe zur Prozessverwaltung

`pid_t fork(void)`
Erzeugen einer Kopie des Prozesses (Child), Parent erhält pid des Kindes zurück oder -1 bei Fehler, Kind erhält 0 als Ergebnis.

`int execve(char* name, char* argv[], char* envp[])`
Überlagern des ausgeführten Programms eines Prozesses (Code, Daten, Stack) durch neues Programm. Andere Varianten:`execl`, `execle`, `execlp`, `execv`, `execvp`

`pid_t getpid(void)`
Rückgabe der eigenen Prozess ID

`pid_t getppid(void)`
Rückgabe der Prozess ID des Elternprozesses.

Notizen

Beispiel UNIX

Hochschule RheinMain

Systemaufrufe zur Prozessverwaltung (Forts.)

`exit(status)`
Beendet den laufenden Prozess und stellt dem Parent den Exit-Status zur Verfügung.

`pid_t wait(int* status)`
Warten auf die Beendigung eines Kindprozesses. Dessen ID wird über den Rückgabewert, sein Status über `status` zurückgegeben.

`pid_t waitpid(pid_t pid, int* status, int opts)`
Warten auf das Ende eines bestimmten Kindprozesses, dessen ID über den Parameter pid übergeben wird.

(wird im Praktikum vertieft)

Notizen

Threads

Hochschule RheinMain

Motivation

Prozesserzeugung, Prozessumschaltung und Prozesskommunikation sind teuer

- → rechenzeitaufwändig zur Laufzeit
- Verluste auch durch Cache-Misses

Wie nutzt man mehrere Prozessoren eines Multiprozessors für eine Applikation?

- Z.B. mehrere kooperierende Prozesse auf verschiedenen Prozessoren
- Muss auf Applikationsebene ausprogrammiert werden !

Wie strukturiert man einen Server-Prozess, der Anforderungen von mehreren Klienten bedienen kann?

- Ein Server-Prozess → keine Parallelität
- Multiplexing für verschiedene Klienten von Hand
  → komplexe Programmierung

Lösung

Einführung von billiger Nebenläufigkeit in einem Prozessadressraum durch „Leichtgewichtsprozesse“, sogenannte **Threads**.

Notizen

Definition

Hochschule RheinMain

Prozess (Einheit der Betriebsmittelverwaltung)

ausführbares Programm, das Code und globale Daten definiert.
privater Adressraum.
Code und Daten über Adressraum zugreifbar.
Menge von Betriebsmitteln

- geöffnete Objekte, Betriebssystem-Objekte wie z.B. Timer, Signale, Semaphore
- dem Prozess durch das Betriebssystem als Folge der Programmausführung zugeordnet.

Menge an Threads

Threads (Aktivitätsträger)

Idee einer „parallel ausgeführten Programmfunktion“
Eigener Prozessor-Context (Registerinhalte usw.)
Eigener Stack (i.d.R. zwei, getrennt für user und kernel mode)
eigener kleiner privater Datenbereich (Thread Local Storage)
Threads eines Prozesses nutzen gemeinsam Programm, Adressraum und alle Betriebsmittel.

Notizen

Schnittstellen

Hochschule RheinMain

Vornehmlich: **POSIX pthreads** (s.u.)

Daneben:

- Java: Klasse Thread in java.lang
  - Z.B. Implementierung der Schnittstelle `java.lang.Runnable` und Implementierung von Methode `run()`.
  - Thread-Modell in JVM implementiert
- C++: Boost Threads verbreitet
- Windows:
  - C-Schnittstelle für Windows API, u.a. `CreateThread(...)`
- LinuxThreads (veraltet)
  - `clone()` System Call
    → Erzeugung eines Prozesses mit Angabe detaillierter Flags, was gemeinsam mit erzeugendem Prozess genutzt werden soll
  - Gehört nicht zum UNIX Standard
    → Programme nicht portierbar

Notizen

POSIX Threads (pthreads)

Hochschule RheinMain

POSIX: Portable Operating System Interface (for unIX)

Familie internationaler Standards ISO/IEC 9945, ursprünglich spezifiziert durch IEEE Computer Society als IEEE 1003

C-API-Spezifikationen → Kompatibilität auf Quellcode-Ebene (kein ABI)

Umfassende Funktionalitäten[2], hier nur Thread Interface („pthreads“) betrachtet.

Auf vielen Systemen verfügbar, insbesondere auch Multiprozessorsystemen, z.B. Linux, MacOS X, FreeBSD, Solaris

Neben C / C++ auch für andere Prgrammiersprachen

Teilweise Bestandteil der libc

ca. 50 Funktionen

[2] POSIX Base Definitions, System Interfaces, and Commands and Utilities (which include POSIX.1, extensions for POSIX.1, Real-time Services, Threads Interface, Real-time Extensions, Security Interface, Network File Access and Network Process-to-Process Communications, User Portability Extensions, Corrections and Extensions, Protection and Control Utilities and Batch System Utilities.

Notizen

Pthreads API

Hochschule RheinMain

API-Aufrufe zum Thread-Management

```
#include <pthread.h>
```

Erzeugen eines Threads

```
int pthread_create(pthread_t * thread,
const pthread_attr_t * attr,
void * (*start_routine)(void *),
void *arg);
```

Sich selbst beenden

```
void pthread_exit(void *retval);
```

Thread-Identifier des aktuellen Threads ermitteln

```
pthread_t pthread_self(void);
```

Warten auf Beendigung eines Threads

```
int pthread_join(pthread_t th, void **thread_return);
```

Beenden eines anderen Threads

```
int pthread_cancel(pthread_t thread);
```


Notizen

Beispiel: Thread Erzeugen/Löschen

Hochschule RheinMain

```
#include <stdio.h>
#include <pthread.h>

void my_thread(int *param);

main(int argc, char *argv[])
{
  pthread_create(&thread,
        NULL,
        (void*)my_thread,
        (void*)&arg);
  .....

  pthread_join(thread, NULL);
  return 0;
}
```

```
void my_thread(int* pcount)
{
  int i;
  int count = *pcount;
  for (i = 0; i < count; i++)
    do_fun();
}
```

Thread erzeugen und starten

Attribute, default falls NULL

Zeiger auf Argument (beliebiger Zeiger)

Warten auf Ende des Thread

Zeiger auf Speicher für Returnwert (NULL falls nicht erwünscht)

Notizen

Kooperationsformen

Hochschule RheinMain

Bei **Einprozessorsystemen** liefern Threads **keinen** Performancegewinn

Dennoch ist die Verwendung von Threads zur Organisation nebenläufiger Prozesse sinnvoll

Beispiel: Server

(a) Verteiler/Arbeiter-Modell

(b) Team-Modell

(b) Fließband-Modell

Notizen

Kooperationsformen

Hochschule RheinMain

Bei **Mehrprozessorsystemen** können Prozesse **nur** mithilfe von Threads und durch nebenläufige Implementierung Performancegewinne erzielen

Notizen

Implementierung von Threads

Hochschule RheinMain

Thread-Bibliothek (*User level threads*)

Thread-Funktionen / Kontextwechsel in Bibliothek auf Applikationsebene implementiert

Betriebssystem kennt nach wie vor nur übliche Prozesse

1:n-Zuordnung: BS „sieht“ nur ganzen Prozess

++ leicht / nachträglich zu implementieren („*retrofit*“)

- - Keine Nutzung von Mehrprozessor-Architekturen

Beispiel: pthreads Implementierung in OSF/DCE

POSIX: „*process scope*“

Notizen

Implementierung von Threads (2)

Hochschule RheinMain

Im BS-Kern (*Kernel level threads*)

Betriebssystem unterstützt Threads

Threads sind die Einheiten, denen Prozessoren zugeordnet werden („*schedulable entities*“).

1:1-Zuordnung (allgem.: m:n)

++ Nutzbarkeit von Mehrprozessor-Architekturen

(- -) Kernel Unterstützung erforderlich

Beispiele: pthreads Implementierung in Linux, , FreeBSD, Solaris, ...

POSIX: „*system scope*“

Notizen

Beispiel: Linux (ab Kernel 2.6)

Hochschule RheinMain

Notizen

***Native POSIX Thread Library* (NPTL)**

Federführung Red Hat

Ziele:

- Konformität zu POSIX Pthreads
- Gute Multiprozessor-Performance
- Niedrige Erzeugungskosten
- Kompatibilität zu LinuxThreads

1:1-Implementierung

- Kernel verwaltet Prozesse
- **pthread_create()** führt zu neuem Prozess unter Nutzung von **clone()**
- Spezielle Kernel-Unterstützung und viel Optimierung im Kernel (z.B. sog. Futexes)

Zusammenfassung

Hochschule RheinMain

Notizen

Was haben wir in Kap. 3 gemacht?

Konzept des sequentiellen Prozesses (Wichtig!).

Strukturierung von Aktivität durch eine Menge von sequentiellen Prozessen, die zueinander nebenläufig ausgeführt werden.

Betriebssystem bietet Anwendungsprogrammierern ein solches Prozesskonzept an der Dienstschnittstelle zur Strukturierung von Anwendungen.

Das Betriebssystem kann Prozesse auch intern zur Strukturierung höherer Funktionalität nutzen.

Ansätze besprochen, wie Betriebssystem das Prozesskonzept implementiert (prinzipiell und speziell am Beispiel UNIX).

Thread-Konzept als performante „Leichtgewichtsprozesse“ vorgestellt.

4. Scheduling

https://advancesystems.ie/why-your-business-needs-to-switch-to-employee-scheduling-software/

Notizen

Scheduling

Einführung
Non-Preemptive Scheduling-Verfahren
Preemptive Scheduling-Verfahren
Scheduling in UNIX
Echtzeit-Scheduling
Zusammenfassung

Notizen

Einführung

Hochschule RheinMain

Grundlegende Begriffe:

Wiederholung aus Kap. 3.1:

- **Scheduler** oder **Dispatcher**: Umschalteinheit
- **Scheduling-Algorithmus**: zugehöriger Algorithmus
- **Prozesswechsel** oder **Kontextwechsel**: Umschaltvorgang, verursacht Kosten.

Non-preemptives Scheduling-Verfahren[1] (*Run-to-Completion*, d.h. Prozess ist solange aktiv, bis er endet oder sich selbst blockiert).

- Non-preemptive-Scheduling-Verfahren sind für General Purpose Systeme mit interaktiven Benutzern nicht geeignet.

Preemptives Scheduling: rechnende Prozesse können suspendiert werden (Prozessorentzug)

- *preemptive-resume*: Fortsetzung ohne Verlust
- *preemptive-repeat*: Beginn von vorne

[1] auch: „kooperatives“ Scheduling-Verfahren

Notizen

Grundlegende Begriffe (2)

Hochschule RheinMain

Prioritäten-basierte Scheduling-Verfahren: ordnen Prozessen Prioritäten zu (relative Wichtigkeit).

Prioritäten können **extern** vorgegeben sein, oder **intern** durch das Betriebssystem selbst bestimmt werden.

Prioritäten können **statisch** sein, d.h. ändern sich während der Bearbeitung nicht, andernfalls: **dynamische** Prioritäten.

Scheduling-Verfahren, die die Prozessorzuteilung auf der Basis gewählter Zeitspannen mittels Uhrunterbrechungen steuern, heißen **Zeitscheiben-basierte** Scheduling-Verfahren.

Mischformen sind möglich

Notizen

Grundlegende Begriffe (3)

Hochschule RheinMain

Mehrstufiges Scheduling[2]: Scheduler auf mehreren Ebenen
Nur die auf einer höheren Ebene durch den entsprechenden Scheduler ausgewählten Prozesse sind dem Scheduler der nächst niederen Ebene für sein Scheduling sichtbar. Scheduling der niederen Ebene wird dann häufig als Dispatching bezeichnet.

Beispiele:

- Virtuelle Maschine (z.B. VirtualBox): Die virtuelle Maschine unterliegt als Anwendungsprozess dem Scheduler des Wirtssystems. In der VM arbeitet ein Gastsystem, dessen Prozesse dessen Scheduler unterliegen.
- 2-stufiges Scheduling in UNIX; Der Scheduler der höheren Ebene transportiert Aufträge vom Hintergrundspeicher in den Arbeitsspeicher und zurück (*Swapping*); der Scheduler der niederen Ebene berücksichtigt ausschließlich Prozesse, die sich im Arbeitsspeicher befinden.

[2] auch: Hierarchisches Scheduling

Notizen

Begriffe aus der Bedientheorie (1)

Hochschule RheinMain

Auftrag: Einheit zur Bearbeitung (z.B. Stapeljob, Dialogschritt)
Bedienzeit:Zeitdauer für die reine Bearbeitung eines Auftrags durch die Bedienstation (hier: den Prozessor)
Wartedisziplin: z.B. (einfache):

- FCFS (*First-Come-First-Served*) oder FIFO (*First-In-First-Out*)
- LIFO (*Last-In-First-Out*)
- Random (zufällige Auswahl)

Ankünfte und Bedienzeiten werden bei Bedienungsmodellen häufig durch stochastische Prozesse modelliert
komplexere Bedienmodelle können mehrere Bedienstationen (Multiprozessorsystem), mehrere Warteräume, mehrphasige Bedienung sowie die Rückführung teilweise bearbeiteter Aufträge enthalten.

Notizen

Begriffe aus der Bedientheorie (2)

Antwortzeit: Zeitdauer vom Eintreffen eines Auftrags bis zur Fertigstellung
Bei Dialogaufträgen Zeitdauer von der Eingabe eines Benutzers (z.B. Drücken der Return-Taste) bis zur Erzeugung einer zugehörigen Ausgabe (z.B. auf dem Bildschirm)
bei Stapelaufträgen auch **Verweilzeit** genannt

Wartezeit: Antwortzeit - Bedienzeit

Durchsatz: Anzahl erledigter Aufträge pro Zeiteinheit

Auslastung: Anteil der Zeit im Zustand „belegt"

Fairness: „Gerechte" Behandlung aller Aufträge, z.B. alle rechenwilligen Prozesse haben gleichen Anteil an der zur Verfügung stehenden Rechenzeit

Notizen

„Guter" Scheduling-Algorithmus

Ziele:

Nutzungsbezogen:

- Kurze Antwortzeiten bei interaktiven Aufträgen.
- Kurze Verweilzeiten für Stapelverarbeitungsaufträge.

Betriebsbezogen:

- Hoher Durchsatz.
- Hohe Auslastung.
- Fairness in der Behandlung aller Aufträge.
- Geringer Aufwand für die Bearbeitung des Scheduling-Algorithmus selbst (Overhead!).

Notizen

Probleme

Hochschule RheinMain

Vorabwissen über die Bedienzeit:

Einige Verfahren verlangen Kenntnis der Bedienzeit eines Auftrags bei dessen Ankunft (vgl. Verfahren zur Maschinenbelegung, Operations Research).

Nur realistisch für wiederkehrende Stapeljobs, nicht realistisch bei interaktiven Aufträgen.

Bei Echtzeit-Scheduling: Annahme / Abschätzung / Formale Bestimmung einer **Oberschranke der Bedienzeit** (*worst case execution time* - WCET) als Voraussetzung für die Echtzeitplanung

Notizen

Moderne Anforderungen (1)

Hochschule RheinMain

Trennung von Strategie und Mechanismus: (Ziel: höhere Flexibilität)

Parametrisierbarer Scheduling-Mechanismus auf niederer Ebene (im Betriebssystemkern).

Parameter können auf höherer Ebene (z.B. über Systemaufrufe in Benutzerprozessen) gesetzt werden, um applikationsbezogene Strategie zu implementieren.

→ Das Scheduling wird damit weiter auf niederer Ebene durchgeführt, aber von der Applikationsebene aus gesteuert.

Beispiel: Ein Datenbank-Management-Prozess kann für seine Kindprozesse, die z.B. bestimmte ihm bekannte Anfragen bearbeiten oder interne Dienste durchführen, deren optimale Einplanung und Abfolge bewirken.

Notizen

Moderne Anforderungen (2)

Hochschule RheinMain

User-Level Scheduler

Über die o.g. Forderung hinaus kann ein Applikationsprogramm dem Betriebssystem z.B. über einen Systemaufruf einen Scheduler übergeben, der für eine Teilmenge der Prozesse deren Scheduling durchführt (Höchstmaß an Flexibilität).

Nur in wenigen Betriebssystemen vorhanden

Mikrokern-Ansatz: Rechenzeit als Ressource, per IPC transferierbar

Gegenstand aktueller Forschung[3]

[3] z.B.:
- Lyons et al: *Scheduling-Context Capabilities*
- Gadepalli et al: *Slite: OS Support for Near Zero-Cost, Configurable Scheduling*

Notizen

Non-Preemptive Scheduling

Hochschule RheinMain

(Annahme: Bekannte Bedienzeiten)

First-Come-First-Served (FCFS)

Shortest-Job-First (SJF)

Prioritäts-Scheduling (Prio)

Notizen

First-Come-First-Served (FCFS)

Algorithmus:

Einfachst-Algorithmus: „Wer zuerst kommt, mahlt zuerst“.
Vollständige Bearbeitung jedes Auftrags, bevor ein neuer begonnen wird.
„Pech“, wenn ein Langläufer vor kurzem Prozess in der Schlange steht

Implementierung:

Die Ready-Queue wird als FIFO- Liste verwaltet.

Bedienmodell:

Notizen

Rechenbeispiel

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt

Resultierender Schedule:

Prozess	Wartezeit	Antwortzeit
1	0	13
2	13	16
3	13+3=16	22

Durchschnittliche Wartezeit:
$(13 + 16)/3 = 29/3$

Im Falle der Ausführungsfolge 3, 2, 1 hätte sich ergeben:
Durchschnittliche Wartezeit: $(6 + 9)/3 = 5$

Notizen

Shortest-Job-First (SJF)

Algorithmus:

Von allen rechenwilligen Prozessen wird derjenige mit der kleinsten Bedienzeitanforderung ausgewählt.

Bei gleicher Bedienzeitanforderung wird nach FCFS gewählt.

Der Algorithmus SJF ist in dem Sinne optimal in der Menge aller möglichen Algorithmen, dass er die kürzeste mittlere Wartezeit für alle Aufträge sichert.

Notwendigkeit der Kenntnis der Bedienzeit!

Notizen

Rechenbeispiel

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt

Resultierender Schedule:

Prozess	Wartezeit	Antwortzeit
1	3+6=9	22
2	0	3
3	3	9

Durchschnittliche Wartezeit:
$(9+3)/3 = 4$

Notizen

Prioritäts-Scheduling (Prio)

Algorithmus:

Jeder Auftrag besitze eine statische Priorität.

Von allen rechenwilligen Prozessen wird derjenige mit der höchsten Priorität ausgewählt.

Bei gleicher Priorität wird nach FCFS ausgewählt.

Notizen

Rechenbeispiel

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit	Priorität
1	13	2
2	3	3
3	6	4

Alle Aufträge seien zur Zeit Null bekannt

Resultierender Schedule:

Prozess	Wartezeit	Antwortzeit
1	6+3=9	22
2	6	9
3	0	6

Durchschnittliche Wartezeit[4]:
$(9+6)/3 = 5$

[4] In diesem Beispiel – abhängig von Prioritätsvergabe sind auch alle anderen Ergebnisse möglich

Notizen

Preemptive Scheduling

(realistisch für heutige Rechensysteme)

- Round-Robin-Scheduling (RR)
- Unterbrechendes Prioritäts-Scheduling
- Mehrschlangen-Scheduling
- Mehrschlangen-Feedback-Scheduling

Notizen

Round-Robin-Scheduling (RR)

Algorithmus:

Menge der rechenwilligen Prozesse linear geordnet.
Jeder rechenwillige Prozess erhält den Prozessor für eine feste Zeitdauer q, die **Zeitscheibe** (*time slice*) oder **Quantum** genannt wird.
Nach Ablauf des Quantums wird der Prozessor entzogen und dem nächsten zugeordnet (preemptive-resume).
Tritt vor Ende des Quantums Blockierung oder Prozessende ein, erfolgt der Prozesswechsel sofort.
Dynamisch eintreffende Aufträge werden z.B. am Ende der Warteschlange eingefügt.

Implementierung:

Die Zeitscheibe wird durch einen Uhr-Interrupt realisiert.
Die Ready Queue wird als lineare Liste verwaltet, bei Ende eines Quantums wird der Prozess am Ende der Ready Queue eingefügt.

Notizen

Round-Robin-Scheduling (RR)

Hochschule RheinMain

Bedienmodell:

Bewertung:

Round-Robin ist einfach und weit verbreitet.
Alle Prozesse werden als gleich wichtig angenommen und fair bedient.
Langläufer benötigen ggf. mehrere „Runden“
Keine Benachteiligung von Kurzläufern
(ohne Bedienzeit vorab zu kennen)
Einziger kritischer Punkt: Wahl der Dauer des Quantums.

- Quantum zu klein → häufige Prozesswechsel, sinnvolle Prozessornutzung sinkt
- Quantum zu groß → schlechte Antwortzeiten bei kurzen interaktiven Aufträgen.

Notizen

Rechenbeispiel

Hochschule RheinMain

Gegeben: Prozessmenge mit 3 Prozessen

Prozess	Bedienzeit
1	13
2	3
3	6

Alle Aufträge seien zur Zeit Null bekannt
Quantum sei $q = 4$

Resultierender Schedule:

Prozess	Wartezeit	Antwortzeit
1	3+4+2=9	22
2	4	7
3	4+3+4=11	17

Durchschnittliche Wartezeit:
$(9 + 4 + 11)/3 = 8$

Notizen

Grenzwertbetrachtung

Hochschule RheinMain

Grenzwertbetrachtung für Quantum q:

$q \to \infty$:
Round-Robin verhält sich wie FCFS.

$q \to 0$:
Round-Robin führt zu sogenanntem **processor sharing**:
jeder der n rechenwilligen Prozesse erfährt $\frac{1}{n}$ der Prozessorleistung.
(Kontextwechselzeiten als Null angenommen).

Notizen

Unterbrechendes Prioritäts-Scheduling

Hochschule RheinMain

Algorithmus:

Jeder Auftrag besitze eine statische Priorität.

Prozesse werden gemäß ihrer Priorität in eine Warteschlange eingereiht.

Von allen rechenwilligen Prozessen wird derjenige mit der höchsten Priorität ausgewählt und bedient.

Wird ein Prozess höherer Priorität rechenwillig (z.B. nach Beendigung einer Blockierung), so wird der laufende Prozess unterbrochen (*preemption*) und in die Ready Queue eingefügt.

Notizen

Mehrschlangen-Scheduling

Hochschule RheinMain

Algorithmus:

Prozesse werden statisch klassifiziert als einer bestimmten Gruppe zugehörig (z.B. interaktiv, batch).

Alle rechenwilligen Prozesse einer bestimmten Klasse werden in einer eigenen Ready Queue verwaltet.

Jede Ready Queue kann ihr eigenes Scheduling-Verfahren haben (z.B. Round-Robin für interaktive Prozesse, FCFS für batch-Prozesse).

Zwischen den Ready Queues wird i.d.R. unterbrechendes Prioritäts-Scheduling angewendet, d.h.: jede Ready Queue besitzt eine feste Priorität im Verhältnis zu den anderen; wird ein Prozess höherer Priorität rechenwillig, wird der laufende Prozess unterbrochen (preemption).

Notizen

Mehrschlangen-Scheduling (2)

Hochschule RheinMain

Bedienmodell (Beispiel):

Notizen

Mehrschlangen-Feedback-Scheduling

Notizen

Prinzip:

Erweiterung des Mehrschlangen-Scheduling.

Rechenwillige Prozesse können im Verlauf in verschiedene Warteschlangen eingeordnet werden (dynamische Prioritäten).

Algorithmen zur **Neubestimmung der Priorität** wesentlich

Bsp. 1: Wenn ein Prozess blockiert, wird die Priorität nach Ende der Blockierung um so größer, je weniger er von seinem Quantum verbraucht hat (Bevorzugung von I/O-intensiven Prozessen).

Bsp. 2: Wenn ein Prozess in einer bestimmten Priorität viel Rechenzeit zugeordnet bekommen hat, wird seine Priorität verschlechtert (Bestrafung von Langläufern).

Bsp. 3: Wenn ein Prozess lange nicht bedient worden ist, wird seine Priorität verbessert (Altern, Vermeidung einer „ewigen“ Bestrafung).

Mehrschlangen-Feedback-Scheduling (2)

Notizen

Bedienmodell:

Bewertung

Hochschule RheinMain

Mit wachsender Bedienzeit sinkt die Priorität, d.h. Kurzläufer werden bevorzugt, Langläufer werden zurückgesetzt.

Wachsende Länge des Quantums mit fallender Priorität verringert die Anzahl der notwendigen Prozesswechsel (Einsparen von Overhead).

Verbesserung der Priorität nach Beendigung einer Blockierung berücksichtigt I/O-Verhalten (Bevorzugung von I/O-intensiven Prozessen). Durch Unterscheidung von Terminal I/O und sonstigem I/O können interaktive Prozesse weiter bevorzugt werden.

sehr flexibel.

Die Scheduler in Windows und Linux arbeiten nach diesem Prinzip

Notizen

Scheduling in Linux (1)

Hochschule RheinMain

Linux 1.2

- Zyklische Liste, Round-Robin

Linux 2.2

- Scheduling-Klassen (Echtzeit, Non-Preemptive, Nicht-Echtzeit)
- Unterstützung für Multiprozessoren

Linux 2.4

- O(n)-Komplexität (jeder Task-Kontrollblock muss angefasst werden)
- Round-Robin
- Teilweiser Ausgleich bei nicht verbrauchter Zeitscheibe
- Insgesamt relativ schwacher Algorithmus

Notizen

Scheduling in Linux (2)

Linux 2.6

- O(1)-Komplexität (konstanter Aufwand für Auswahl unabhängig von Anzahl Tasks)
- Run Queue je Priorität
- Zahlreiche Heuristiken für Entscheidung I/O-intensiv oder rechenintensiv
- Sehr viel Code

ab Linux Kernel 2.6.23: „Completely Fair Scheduler“ (CFS)

- Sehr gute Approximation von Processor Sharing
- Task mit geringster *Virtual Runtime* (größter Rückstand) bekommt Prozessor
- Zeit-geordnete spezielle Baumstruktur für Taskverwaltung ($\rightarrow$ O(log n)-Komplexität)
- Kein periodischer Timer-Interrupt sondern One-Shot-Timer („tickless Kernel“)

Notizen

Echtzeit-Scheduling

Scheduling in Realzeit-Systemen beinhaltet zahlreiche neue Aspekte. Hier nur erster kleiner Einblick.[5]

Varianten in der Vorgehensweise

- *Statisches Scheduling:*
  Alle Daten für die Planung sind vorab bekannt, die Planung erfolgt durch eine Offline-Analyse.
- *Dynamisches Scheduling:*
  Daten für die Planung fallen zur Laufzeit an und müssen zur Laufzeit verarbeitet werden.
- *Explizite Planung:*
  Dem Rechensystem wird ein vollständiger Ausführungsplan (Schedule) übergeben und zur Laufzeit befolgt (Umfang kann extrem groß werden).
- *Implizite Planung:*
  Dem Rechensystem werden nur die Planungsregeln übergeben.

[5] Mehr dazu im Listenfach „Echtzeitverarbeitung“ im nächsten SoSe

Notizen

Klassifizierung

Notizen

Periodische Prozesse

Gewisse Prozesse müssen häufig zyklisch, bzw perodisch ausgeführt werden.

Hard-Realtime-Prozesse müssen unter allen Umständen ausgeführt werden (ansonsten sind z.B. Menschenleben bedroht).

Zeitliche **Fristen (*Deadlines*)** vorgegeben, zu denen der Auftrag erledigt sein muss.

Scheduler muss die Erledigung aller Hard-Realtime-Prozesse innerhalb der Fristen **garantieren**.

Scheduling geschieht in manchen Anwendungssystemen statisch vor Beginn der Laufzeit (z.B. Automotive). Dazu muss die Bedienzeit-Anforderung (z.B. worst case) bekannt sein.

Im Falle von dynamischem Scheduling sind das **Rate-Monotonic** (RMS) und das **Earliest-Deadline-First** (EDF) Scheduling-Verfahren verbreitet.

Notizen

Rate Monotonic Scheduling (RMS) (1)

Hochschule RheinMain

Ausgangspunkt: Periodisches Prozessmodell

- Planungsproblem gegeben als Menge unterbrechbarer, periodischer Prozesse P_i mit Periodendauern Δp_i und Bedienzeiten Δe_i.
- Perioden zugleich Fristen.

RMS ordnet Prozessen **feste Prioritäten** proportional zur **Rate**[6] zu:

$$prio(i) < prio(j) \iff \frac{1}{\Delta p_i} < \frac{1}{\Delta p_j}$$

Daher auch *fixed priority scheduling*

Die meisten Echtzeit-Betriebssysteme unterstützen prioritätsbasiertes, unterbrechendes Scheduling

→ Voraussetzungen für die Anwendung sind unmittelbar gegeben

Zur Festlegung der Prioritäten genügt allein die Kenntnis der Periodendauern Δp_i

[6] = Kehrwert der Periodendauer

Notizen

Rate Monotonic Scheduling (RMS) (2)

Hochschule RheinMain

RMS Zulassungskriterium (*admission test*):
Wenn für n periodische Prozesse gilt ...:

$$\sum_{i=0}^{n} \frac{\Delta e_i}{\Delta p_i} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right)$$

...dann ist bei Prioritätsvergabe nach RMS **garantiert**, dass alle Fristen eingehalten werden.

Hinreichendes (nicht: notwendiges) Kriterium

Einfach zu überprüfen, mathematisch beweisbare Garantie

Erfordert Kenntnis der *worst case* Bedienzeiten (WCET)

Notizen

Beispiel

Hochschule RheinMain

Gegeben: Prozessmenge mit 2 Prozessen

Prozess i	Bedienzeit Δe_i	Periode Δp_i
1	3	7
2	2	5

$$\frac{3}{7} + \frac{2}{5} \approx 0,8286$$

$$2 \cdot \left(2^{\frac{1}{2}} - 1\right) \approx 0,8284$$

→ Kriterium knapp nicht erfüllt, trotzdem wurde ein Plan gefunden

Aus RMS resultierender Schedule [7]:

[7] (wg. $\frac{1}{7} < \frac{1}{5}$ bekommt P_2 höhere Priorität)

Notizen

Earliest Deadline First Scheduling (EDF) (1)

Hochschule RheinMain

Strategie: *Earliest Deadline First* (EDF)

- Der Prozessor wird demjenigen Prozess P_i zugeteilt, dessen Frist d_i den kleinsten Wert hat (am nächsten ist)
- Wenn es keinen rechenbereiten Prozess gibt, bleibt der Prozessor untätig (d.h. „idle“)

Falls EDF keinen brauchbaren Plan liefert, gibt es keinen (!)

Zur Planung nach EDF genügt allein die Kenntnis der Fristen, bzw. der Periodendauern Δp_i

Die Umsetzung eines EDF-Planes mithilfe des prioritätsbasierten, unterbrechenden Scheduling erfordert die dynamische Änderung von Prozessprioritäten zur Laufzeit.

Daher auch *dynamic priority scheduling*

Nicht alle Echtzeitbetriebssysteme unterstützen dynamische Prioritäten.

Notizen

Earliest Deadline First Scheduling (EDF) (2)

Hochschule RheinMain

EDF Zulassungskriterium (*admission test*):
Wenn für n periodische Prozesse gilt ...:

$$\sum_{i=0}^{n} \frac{\Delta e_i}{\Delta p_i} \leq 1$$

...dann ist bei Planung nach EDF **garantiert**, dass alle Fristen eingehalten werden.

Notwendiges und hinreichendes Kriterium

Einfach zu überprüfen, mathematisch beweisbare Garantie

Erfordert Kenntnis der *worst case* Bedienzeiten (WCET)

Notizen

Beispiel

Hochschule RheinMain

Gegeben: Prozessmenge mit 2 Prozessen

Prozess i	Bedienzeit Δe_i	Periode Δp_i
1	4	7
2	2	5

$$\frac{4}{7} + \frac{2}{5} \approx 0,97143 < 1$$

→ Kriterium erfüllt, Plan existiert

Aus EDF resultierender Schedule

Notizen

Gegenüberstellung RMS ↔ EDF

Hochschule RheinMain

RMS

- - Keine 100% Auslastung möglich
++ Auf gängigen Echtzeit-BS direkt einsetzbar
++ Bei Überlastsituationen werden zunächst niedrig priorisierte Prozesse nicht mehr bedient

EDF

++ 100% Auslastung möglich
- - Erfordert dynamische Prioritäten - nicht auf allen Echtzeit-BS möglich
- - Bei Überlastsituationen erratisches Verhalten

Notizen

Zusammenfassung

Hochschule RheinMain

Was haben wir in Kap. 4 gemacht?

Scheduling-Verfahren

- Kenngrößen wie Antwortzeit, Auslastung und Fairness relevant
- einfache Verfahren: FCFS, SJF, PRIO
- unterbrechende Verfahren Round-Robin, statische Prioritäten, Mehrschlangen-Verfahren
- flexible Feedback-Algorithmen, nach denen Prioritäten dynamisch neu berechnet werden
- Beispiel Linux
- Grundzüge des Echtzeit-Scheduling: RMS und EDF

Notizen

5. Prozesssynchronisation

https://www.animierte-gifs.net/data/media/1798/animiertes-richter-bild-0001.gif

Notizen

Prozesssynchronisation

Einführung
Synchronisationsprimitive
Klassische Synchronisationsprobleme
Zusammenfassung

Notizen

Einführung

Hochschule RheinMain

Problemstellung

Gegeben: Nebenläufige (konkurrente) Prozesse (vgl. Kap. 3, engl.: *concurrent processes*).

Ziel: Vermeidung ungewollter gegenseitiger Beeinflussung.

Ziel: Unterstützung gewollter Kooperation zwischen Prozessen:

- Gemeinsame Benutzung von Betriebsmitteln (*sharing*)
- Übermittlung von Signalen
- Nachrichtenaustausch

⇒ **Fazit: Mechanismen zur Synchronisation und Kommunikation von Prozessen sind notwendig.**

Notizen

Beispiel

Hochschule RheinMain

Gemeinsame Benutzung eines Speicherbereiches (Hier: Datum „Kontostand“)

Ungewollte gegenseitige Beeinflussung

Gegeben: 2 Prozesse:

Prozess 1:

```
/* Gehaltsueberweisung */
z = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
```

Prozess 2:

```
/* Dauerauftrag Miete */
x = lies_kontostand();
x = x - 800;
schreibe_kontostand(x);
```


Notizen

Möglicher Ablauf

Hochschule RheinMain

Prozess 1:

```
/* Gehaltsueberweisung */
z = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
```

Prozess 2:

```
/* Dauerauftrag Miete */
x = lies_kontostand();
x = x - 800;
schreibe_kontostand(x);
```

Mögliche Ausführungsreihenfolge der Anweisungen in Prozess 1 und 2:

```
/* Gehaltsueberweisung */   /* Dauerauftrag Miete */
z = lies_kontostand();
                            x = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
.                           x = x - 800;
.                           schreibe_kontostand(x);
```

Pech, die Gehaltsüberweisung ist „verloren gegangen“ :-(

Bei anderen Reihenfolgen werden die beiden Berechnungen „richtig“ ausgeführt, oder es geht der Dauerauftrag verloren.

Notizen

Definitionen (1)

Hochschule RheinMain

Annahme: Prozesse mit Lese/Schreib-Operationen auf Betriebsmitteln

Definition: Prozesskonflikt

Zwei nebenläufige Prozesse heißen **im Konflikt zueinander stehend** oder **überlappend**, wenn es ein Betriebsmittel gibt, das sie gemeinsam (lesend und schreibend) benutzen, ansonsten heißen sie **unabhängig** oder **disjunkt**.

Definition: Zeitkritische Abläufe (*race conditions*)

Folgen von Lese/Schreib-Operationen der verschiedenen Prozesse heißen **zeitkritische Abläufe** (engl. ***race conditions***), wenn die Endzustände der Betriebsmittel (Endergebnisse der Datenbereiche) abhängig von der zeitlichen Reihenfolge der Lese/Schreib-Operationen sind.

Notizen

Definitionen (2)

Hochschule RheinMain

Definition: Wechselseitiger Ausschluss (*mutual exclusion*)

Ein Verfahren, das verhindert, dass zu einem Zeitpunkt mehr als ein Prozess auf ein gemeinsames Datum zugreift, heisst Verfahren zum **wechselseitigen Ausschluss** (engl. ***mutual exclusion***):

Bemerkung: Ein Verfahren zum wechselseitigen Ausschluss vermeidet zeitkritische Abläufe. Es löst damit ein Basisproblem des Concurrent Programming.

Definition: Kritischer Abschnitt (*critical section*)

Der Teil eines Programms, in dem auf gemeinsam benutzte Datenbereiche zugegriffen wird, heisst **kritischer Abschnitt** oder **kritischer Bereich** (engl. ***critical section*** oder **critical region**):

Bemerkung: Ein Verfahren, das sicherstellt, dass sich zu keinem Zeitpunkt zwei Prozesse in ihrem kritischen Abschnitt befinden, vermeidet zeitkritische Abläufe.
Kritische Abschnitte realisieren sog. komplexe **unteilbare** oder **atomare** Operationen.

Notizen

(Nicht-)atomare Operationen (1)

Hochschule RheinMain

```
/* Gehaltsüberweisung */
z = lies_kontostand();
z = z + 1000;
schreibe_kontostand(z);
```

Nicht Atomar

```
/* Ueberweisung 2.0 */
int kontostand;

kontostand += 1000;
```

Atomar?

→ entspr. Maschinencode:

```
MOV kontostand,EAX
ADD #1000,EAX
MOV EAX,kontostand
```

3 Instruktionen
→ Nicht Atomar!

Auch Operationen, die in Hochsprache (hier: C) atomar zu sein scheinen, sind es auf Maschinenebene u.U. nicht!

Teilweise architekturabhängig

Notizen

(Nicht-)atomare Operationen (2)

Hochschule RheinMain

Selbst einzelne Maschineninstruktionen können u.U. nicht-atomar sein!

Beispiel

Assemblercode

```
LA r2,0x1001
SW 0x12345678,(r2)
```

1 Maschinenbefehl → Atomar?

Im Speicher:

0x1000		0x12	0x34	0x56
0x1004	0x78			
	0	1	2	3

Ungerade Adresse → Mehrere Buszyklen zum Schreiben eines Datenwortes erforderlich → unterbrechbar!

Notizen

Ein „guter“ Algorithmus

Hochschule RheinMain

Forderungen an einen guten Algorithmus zum wechselseitigen Ausschluss:

Zu jedem Zeitpunkt darf sich nur ein Prozess in seinem kritischen Abschnitt befinden (Korrektheit, Basisforderung).

Es dürfen keine Annahmen über die Ausführungsgeschwindigkeiten oder die Anzahl der unterliegenden Prozessoren gemacht werden.

Kein Prozess, der sich nicht in seinem kritischen Abschnitt befindet, darf andere Prozesse blockieren (Fortschritt).

Alle Prozesse werden gleich behandelt (Fairness).

Kein Prozess darf unendlich lange warten müssen, bis er in seinen kritischen Abschnitt eintreten kann
(→ „Verhungern“, engl. *starvation*).

Notizen

Synchronisationsprimitive

Hochschule RheinMain

Notizen

Gliederung

- Wechselseitiger Ausschluss mit aktivem Warten
- Wechselseitiger Ausschluss mit passivem Warten
- Semaphore
- Höhere Synchronisationsprimitive

Wechselseit. Ausschluss mit aktivem Warten

Hochschule RheinMain

Notizen

Generelle Vorgehensweise

```
enter_crit(); /* Prolog */
/* critical section */
 <statement>
      …
 <statement>
leave_crit(); /* Epilog */
```

Kritische Sektion

- Prolog/Epilog-Paar
- **Aktives Warten** auf Eintritt in den kritischen Abschnitt (engl. *busy waiting*).
- Aktives Warten für einen längeren Zeitraum verschwendet Prozessorzeit.
- Alle Prozesse müssen sich an das Vorgehen halten.

Lösungsansätze

Hochschule RheinMain

Sperren aller Unterbrechungen

Sperrvariablen

Striktes Alternieren

Lösung von Peterson[1]

Atomare read-modify-write Instruktionen[1]

[1] brauchbar

Notizen

1. Sperren aller Unterbrechungen

Hochschule RheinMain

Einfachste Lösung, Vorgehen:

Jeder Prozess **sperrt** vor Eintritt in seinen kritischen Abschnitt alle **Unterbrechungen** (*disable interrupts*).
Jeder Prozess **lässt** Unterbrechungen am Ende seines kritischen Abschnitts **wieder zu** (*enable interrupts*).

Funktionsweise:

Der Prozessor kann nur dann zu einem anderen Prozess wechseln, wenn eine Unterbrechung auftritt. Also ist für diese Lösung die **Korrektheitsforderung erfüllt**.

Bemerkungen:

Lösung ist **unbrauchbar für allgemeine Anwendungsprozesse**, da:

- das Sperren/Zulassen von Unterbrechungen i.d.R. ein privilegierter Maschinenbefehl ist
- nicht zugesichert werden kann, dass Anwendungen die Interrupts auch wieder zulassen (z.B. wegen Programmierfehler).

Lösung wird häufig **innerhalb des Betriebssystemkerns** eingesetzt, um wechselseitigen Ausschluss zwischen Kernroutinen (z.B. mit einem Interrupt-Handler) zu gewährleisten.
Lösung ist auch **unbrauchbar** im Falle eines **Multiprozessor**-Systems, da sich die Interrupt-Sperre i.d.R. nur auf einen Prozessor auswirkt.

Notizen

2. Sperrvariablen

Hochschule RheinMain

Einfacher, aber falscher Lösungsansatz:

Jedem kritischen Abschnitt wird eine **Sperrvariable** zugeordnet:

- Wert = 0 bedeutet „frei“,
- Wert = 1 bedeutet „belegt“.

Jeder Prozess prüft die Sperrvariable vor Eintritt in den kritischen Abschnitt:

- Ist sie 0, so setzt er sie auf 1 und tritt in den kritischen Abschnitt ein.
- Ist sie 1, so wartet er, bis sie den Wert 0 annimmt.

Am Ende seines kritischen Abschnitts setzt der Prozess den Wert der Sperrvariablen auf 0 zurück.

Prozess 1:

```
/* Prolog */
while(sperrvar) { ; }
sperrvar = 1;
/* kritischer Bereich */
...
/* Epilog */
sperrvar = 0;
```

Prozess 2:

```
 Prolog */
while(sperrvar) { ; }
sperrvar = 1;
/* kritischer Bereich */
...
/* Epilog */
sperrvar = 0;
```


Notizen

2. Sperrvariablen (2)

Hochschule RheinMain

Bemerkungen:

Prinzipiell der gleiche Fehler wie beim Konto-Beispiel:
Zwischen Abfrage der Sperrvariablen und folgendem Setzen kann der Prozess unterbrochen werden.
Damit ist es möglich, dass sich beide Prozesse im kritischen Abschnitt befinden (→**Korrektheitsbedingung verletzt !**).
Speicher (-wörter, -variablen) erlauben i.d.R. nur unteilbare Lese-, bzw. Schreibzugriffe, jedoch kein unteilbares *read-modify-write* (Eigenschaft der Architektur des Speicherwerks).

```
/* Prolog */                  /* Prolog */
while(sperrvar) { ; }         while(sperrvar) { ; }
...                           sperrvar = 1;
sperrvar = 1;                 ...
/* kritischer Bereich */      /* kritischer Bereich */
...                           ...
/* Epilog */                  /* Epilog */
sperrvar = 0;                 sperrvar = 0;
```


Notizen

3. Striktes Alternieren

```
int dran = 0; /* gem. Variable: wer ist "dran" ? */
```

Prozess 0:

```
/* Prolog */
while(dran != 0) { ; }
/* kritischer Bereich */
...
/* Epilog */
dran = 1;
```

Prozess 1:

```
/* Prolog */
while(dran != 1) { ; }
/* kritischer Bereich */
...
/* Epilog */
dran = 0;
```

Gemeinsame Variable `dran` gibt an, welcher Prozess[2] den kritischen Bereich betreten darf

Prozesse wechseln sich ab: 0,1,0,1,...

Die Lösung erfüllt die Korrektheitsbedingung, aber

→ Fortschrittsbedingung (3) kann verletzt sein, wenn ein Prozess wesentlich langsamer als der andere ist.

Fazit: **keine ernsthafte Lösung**.

[2] Hier: 0 oder 1

Notizen

4. Lösung von Peterson

Historische Vorläufer:

- Anfang der 60er Jahre viele Lösungsansätze, nur wenige erfüllten alle genannten Bedingungen
- Erste korrekte Software-Lösung für 2 Prozesse: Algorithmus von Dekker

Neue Lösung zum wechselseitigen Ausschluss:

- Lösung von Peterson (1981) (im folgenden betrachtet)
- Basiert (ebenfalls) auf unteilbaren Speicheroperationen und aktivem Warten, ist aber einfacher
- Prolog: `enter_crit()`, Epilog: `leave_crit()`

Weitere Lösungen für mehr als zwei Prozesse von:

- Dijkstra, Peterson, Knuth, Eisenberg/McGuire, Lamport (hier nicht weiter diskutiert)

Notizen

Peterson-Algoritmus (1)

Hochschule RheinMain

```
#define N    2                  /* Anzahl der Prozesse          */

/* gemeinsame Variablen                                          */
volatile int dran;              /* Wer kommt dran?              */
volatile bool interested[N];    /* Wer will, anfangs alle false */

/* Prolog-Operation, vor Eintritt in den
   kritischen Bereich ausfuehren   */
void enter_crit(int process) { /* process: wer tritt ein: 0,1  */
   int other;                   /* Nummer des anderen Prozesses */
   other = 1 - process;
   interested[process] = true; /* zeige eigenes Interesse       */
   dran = process;              /* setze Marke, unteilbar!      */
   while(dran==process && interested[other]);
                                /* ev. Aktives Warten !!!       */
}
```

```
/* Epilog-Operation, nach Austritt aus dem
   krit. Bereich ausfuehren */
void leave_crit(int process) { /* process: wer verlaesst: 0,1  */
   interested[process] = false; /* Verlasse krit. Bereich      */
}
```


Notizen

Peterson-Algorithmus (2)

Hochschule RheinMain

Anfangszustand

intrst[]:	0	0
dran:	0	

Notizen

Peterson-Algorithmus (3)

Prozess 1 betritt kritischen Abschnitt, Prozess 2 nicht

intrst[]:	1	0
dran:	0	

Notizen

Peterson-Algorithmus (4)

Prozess 1 ist in kritischem Abschnitt, Prozess 2 versucht, ihn zu betreten

intrst[]:	1	1
dran:	1	

Prozess 2 wartet, bis intrst[0] = 0

Notizen

Peterson-Algorithmus (5)

Kritischer Fall:
beide versuchen gleichzeitig,
den kritischen Abschnitt
zu betreten

intrst[]: 1 1

dran: 0 oder 1

Schreibzugriff auf „dran" ist atomar. Nur einer von beiden kann „dran" erfolgreich setzen.

Wer Erfolg hat, muss hier warten!

Notizen

5. Atomare read-modify-write Instruktionen

Algorithmen, die nur auf atomaren Speicherzugriffen basieren sind

- Komplex → fehlerträchtig
- Auf eine feste Teilnehmerzahl beschränkt

Zudem besteht bei einigen Verfahren die Gefahr des „Verhungerns" (engl. *starvation*)

→ Notwendigkeit einer effizienten Lösung

Lösung durch Hardware-Unterstützung:

Einführung spezieller Maschinenbefehls: unteilbares Lesen einer Speicherzelle mit anschließendem Schreiben eines Wertes in die Zelle (Speicherbus wird zwischendurch **nicht** freigegeben)

- Heute Standard auf praktisch allen Architekturen
- Notwendig für Mehrprozessor-Systeme
- Häufig auch zusätzliche unteilbare Maschinen-Operationen zur Listenmanipulation (z.B. für Ready-Queue)
- Oder LOCK/UNLOCK-Paare, um kurze Folgen beliebiger Instruktionen atomar auszuführen (z.B. i860).

Notizen

Beispiel: Test-and-Set-Befehl (1)

Hochschule RheinMain

Jedem kritischen Abschnitt wird eine Sperrvariable `flag` zugeordnet
Wert = 0 bedeutet „frei“, Wert = 1 bedeutet „belegt“
Auf Test-and-Set basierende Sperrvariablen mit aktivem Warten heißen auch *spin locks*
Große Bedeutung in Multiprozessor-Betriebssystemen
Typische Assembler-Routinen für Prolog und Epilog mit Test-and-Set-Instruktion „TAS“:

```
enter_crit:
  tas reg,flag     | kopiere flag in reg, setze flag = 1
  cmp reg,#0       | war flag vorher 0?
  jnz enter_crit   | nein -> Sperre war belegt -> warten
  ret              | ja -> darf einreten, zurueck zum Aufrufer

leave_crit:
  mov flag,#0      | loesche flag -> setze Sperre zurueck
  ret              | zurueck zum Aufrufer
```

(Im Gegensatz zur Darstellung hier erfolgt die Realisierung i.d.R. über Makros, da Prozeduraufrufe zur Laufzeit zu großen Overhead darstellen)

Notizen

Lock Holder Preemption Problem

Hochschule RheinMain

Problem der *Lock Holder Preemption* tritt in Verbindung mit Spinlocks bei Mehrprozessor-
Systemen, sowie bei Virtualisierung auch bei Einprozessorsytemen auf.

Szenario:

- Ein (ggf. virtueller) Prozessor hat einen durch Spinlock geschützten kritischen Bereich betreten und wird in diesem Zustand unterbrochen.
- Ein anderer (virtueller) Prozessor der versucht, den kritischen Bereich zu betreten, muss **aktiv** auf Freigabe des Spinlocks warten.
- Das kann sehr lange dauern (Quantum)

http://www-archive.xenproject.org/files/xensummitboston08/LHP.pdf

Notizen

Wechselseit. Ausschluss mit passivem Warten

Hochschule RheinMain

Bisher

Prolog-Operationen zum Betreten des kritischen Abschnitts führen zum **aktiven Warten**, bis der betreffende Prozess in den kritischen Abschnitt eintreten kann.
Der wartende Prozess **gibt den Prozessor dabei nicht ab**.

→ verbraucht viel Rechenzeit
→ hindert ggf. andere Prozesse an der Freigabe des kritischen Abschnittes, auf die er selbst wartet.

Lediglich auf Multiprozessor-Systemen kann kurzzeitiges aktives Warten zur Vermeidung eines Prozesswechsels sinnvoll sein (spin locks).

Ziel: Vermeidung von verschwendeter Prozessorzeit. Vorgehensweise:

Prozesse **blockieren**, wenn sie nicht in ihren kritischen Abschnitt eintreten können.
Ein blockierter Prozess wird **durch den** aus dem kritischen Abschnitt **austretenden Prozess entblockiert**.

Notizen

Passives Warten: Einfachste Primitive

Hochschule RheinMain

Einfachste Primitive werden vielfach als `SLEEP`[3] und `WAKEUP` bezeichnet.

`SLEEP()` blockiert den ausführenden Prozess, bis er von einem anderen Prozess geweckt wird.

`WAKEUP(process)` weckt den Prozess `process`. Der ausführende Prozess wird dabei nie blockiert.

Häufig wird als Parameter von `SLEEP` und `WAKEUP` ein Ereignis (Speicheradresse einer beschreibenden Struktur) verwendet, um die Zuordnung treffen zu können, (vgl. 3.2 Beispiel UNIX(7) Kernroutinen).

Diese Primitive können auch der allgemeinen ereignisorientierten Kommunikation dienen.

[3] Achtung: nicht zu verwechseln mit C-Funktion `sleep()`

Notizen

Mutex-Locks

Hochschule RheinMain

Der Begriff **Mutex** ist von *mutual exclusion* (= Wechselseitiger Ausschluss) abgeleitet.

ein Mutex bietet folgende Funktionen:

- *lock* als Prolog-Operation zum Betreten des kritischen Abschnitts
- *unlock* als Epilog-Operation beim Verlassen des kritischen Abschnitts

Mutexe können als Spezialfall von Semaphoren angesehen werden (vgl. 5.2.3).

Bei einigen Implementierungen entblockiert unlock **alle** wartenden Prozesse, damit sich diese dann erneut um das Betreten des kritischen Abschnitts bewerben.

Notizen

Beispiel: Pthreads - API: Mutexe

Hochschule RheinMain

```
#include <pthread.h>
```

Anlegen einer Mutex-Variablen (mehrere Varianten):

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
```

Initialisieren einer Mutex-Variablen:

```
pthread_mutex_init(pthread_mutex_t *mutex,
                   const pthread_mutexattr_t *mutexattr);
```

Lock anfordern:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Lock anfordern, falls ohne Blockieren möglich:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Lock freigeben:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Lock zerstören:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```


Notizen

Problem der Prioritätsinversion

Problem der Prioritätsinversion tritt bei prioritätsbasiertem Scheduling unterbrechbarer Prozesse in Zusammenhang mit kritischen Abschnitten auf

Szenario:

- Prozess H hat hohe Priorität, Prozess L hat niedere Priorität. Scheduling-Regel: wenn H rechenwillig ist, soll H ausgeführt werden
- H wird rechenwillig, während L sich in seinem kritischen Abschnitt befindet. H will ebenfalls in seinen kritischen Abschnitt eintreten
- Der höher priore Prozess H kann nicht weiter ausgeführt werden, da L den kritischen Abschnitt noch nicht freigegeben hat
- Die Dauer der Verzögerung hängt nicht nur von L ab, sondern von allen Prozessen, die aufgrund ihrer Priorität vor H ausgeführt werden
- Insbesondere kann ein Prozess M, dessen Priorität über der von L liegt, den krit. Abschnitt beliebig lange blockieren, ohne selbst daran beteiligt zu sein.

Es muss also der nieder priore Prozess ausgeführt werden!
Diese Situation heißt Prioritätsinversionsproblem.

Notizen

Problem der Prioritätsinversion (2)

Notizen

Semaphore

1965 von Edsger W. Dijkstra eingeführt

Supermarkt-Analogie:

- Kunde darf den Laden nur mit einem Einkaufswagen betreten
- Es steht nur eine bestimmte Anzahl von Einkaufswagen bereit
- Sind alle Wagen vergeben, müssen neue Kunden warten, bis ein Wagen zurückgegeben wird

Semaphor besteht aus:

- einer **Zählvariablen**, die begrenzt, wieviele Prozesse augenblicklich ohne Blockierung passieren dürfen
- einer **Warteschlange** für (passiv) wartende Prozesse

Notizen

Operationen auf Semaphoren (1)

Initialisierung

- Zähler auf initialen Wert setzen
- „Anzahl der freien Einkaufswagen“

Operation P(): Passier(-Wunsch) (auch: **DOWN()**)

- Zähler $= 0 \rightarrow$ Prozess in Warteschlange setzen, blockieren
- Zähler > 0: Prozess kann passieren
- In beiden Fällen wird der Zähler (ggf. nach dem Ende der Blockierung) erniedrigt
- P steht für „proberen” (niederländisch für „testen”)

Operation V(): Freigeben (auch: **UP()**)

- Zähler wird erhöht
- Falls es Prozesse in der Warteschlage gibt, wird einer de-blockiert (und erniedrigt den Zähler dann wieder, s.o.)
- V steht für „verhogen” (niederländisch für „erhöhen”)

Notizen

Operationen auf Semaphoren (2)

Hochschule RheinMain

Bemerkungen

P() und V()sind atomare Operationen:

- Das Überprüfen, Dekrementieren und Sich-Schlafen-Legen des Aufrufers bei P() sowie das inkrementieren und ggf. Wecken eines Prozesses bei V() ist jeweils eine Transaktion, die nur vollständig „in einem Zuge" ausgeführt werden kann

Kein Prozess wird bei der Ausführung der V()-Operation blockiert

Notizen

Implementierung von Semaphoren

Hochschule RheinMain

I.d.R. als Systemaufrufe

Intern Nutzung für Sich-Schlafen-Legen und Aufwecken

Wesentlich ist die Unteilbarkeit von P() und V():

- Auf Einprozessorsystemen: durch Sperren aller Unterbrechungen während der Ausführung von P() und V(). Zulässig, da nur wenige Maschineninstruktionen zur Implementierung nötig sind, und da dem Betriebssystemkern vertraut wird.
- Auf Multiprozessorsystemen: Jedem Semaphor wird ein Spinlock mit aktivem Warten vorgeschaltet. So kann stets höchstens ein Prozessor den Semaphor manipulieren
- Man beachte: Unterscheide zwischen aktivem Warten auf den Zugang zum Semaphor, der einen kritischen Abschnitt schützt (einige Instruktionen, Mikrosekunden) und Aktivem Warten auf den Zugang zum kritischen Abschnitt selbst (problemabhängig, Zeitdauer nicht vorab bekannt oder begrenzt)

Notizen

Binärsemaphore und Zählsemaphore

Semaphore, die nur die Werte 0 und 1 annehmen, heißen *binäre Semaphore*

ansonsten heißen sie *Zählsemaphore*

Binäre Semaphore dienen zur Realisierung des wechselseitigen Ausschlusses

Ein mit $n > 1$ initialisierter Zählsemaphor kann zur Kontrolle der Benutzung eines in n Exemplaren vorhandenen Typs von sequentiell wiederverwendbaren Betriebsmitteln verwendet werden

Semaphore sind weit verbreitet, innerhalb von Betriebssystemen und zur Programmierung nebenläufiger Anwendungen

Programmierung mit Semaphoren ist oft fehleranfällig, wenn mehrere Semaphore benutzt werden müssen (vgl. 5.3 und Kap. 7 (Deadlocks))

Mutex-locks (vgl. 5.2.2) können als binäre Semaphore angesehen werden.

Notizen

Anwendung: Wechselseitiger Ausschluss

```
semaphore sem = 1;   /* Init. mit 1          */
...                  /* -> binaere Semaphore */
P(sem);              /*   Prolog             */
     <statement> ;   /* krit. Abschnitt      */
     ...
     <statement> ;   /* krit. Abschnitt      */
V(sem);              /*   Epilog             */
...
```

Der mit 1 initialisierte Semaphor `sem` wird von allen beteiligten Prozessen benutzt

Jeder Prozess klammert seinen kritischen Abschnitt mit `P(sem)` zum Eintreten und `V(sem)` zum Verlassen

Notizen

Anwendung: Betriebsmittelvergabe

Hochschule RheinMain

Es seien `n` Exemplare vom Betriebsmitteltyp `bm` vorhanden

Jedes BM dieses Typs sei sequentiell benutzbar

```
semaphore bm = n;

P(bm);        /*   Beantragen    */

    Benutze das Betriebsmittel

V(bm);        /*   Freigeben     */
```

Dem BM-Typ `bm` wird ein Semaphor `bm`, zugeordnet

Beantragen eines BM durch `P(bm)`, evtl. blockieren, bis ein BM verfügbar wird

Nach der Benutzung freigeben des BM durch `V(bm)`
(Es kann damit von einem weiteren Prozess benutzt werden)

Notizen

Beispiel: Erzeuger/Verbraucher-Problem

Hochschule RheinMain

Gegeben:

- Datenobjekte (z.B. Nachrichten) fester Größe
- Pufferspeicher von begrenzter Größe (n Objekte)

Aufgabe

- Verschiedene Prozesse legen Daten mit Hilfe einer Funktion `insert()` im Puffer ab („Erzeuger"), bzw. entnehmen sie mit einer Funktion `remove()` („Verbraucher")
- `remove()` soll warten, wenn keine Objekte im Puffer sind
- `insert()` soll warten, wenn kein Platz mehr im Puffer ist

Notizen

Lösung mit Semaphoren

```
/* Semaphore initialisieren:
 * empty = Puffergroesse,
 * full=0, mutex=1
 */

void insert(int item) {
    P(&empty);
    P(&mutex);
/* hier: item in Puffer
   stellen              */
    V(&mutex);
    V(&full);
}

int remove(void) {
    P(&full);
    P(&mutex);
/* hier: vorderstes Item
   aus Puffer holen    */
    V(&mutex);
    V(&empty);
}
```

full: zählt belegte Einträge im Puffer, verhindert Entnahme aus leerem Puffer

empty: verwaltet freie Plätze im Puffer, verhindert Einfügen in vollen Puffer

mutex: schützt den kritischen Bereich vor gleichzeitigem Betreten (binär-Semaphor: nimmt nur Werte 1/0 an)

Notizen

Anwendung: Verstecken von Interrupts

Einem E/A-Gerät wird ein Geräteprozess und ein mit 0 initialisierter Semaphor zugeordnet

Nach dem Start des Geräteprozesses führt dieser eine P-Operation auf dem Semaphor aus (und blockiert)

Im Falle eines Interrupts des Gerätes führt der Interrupt-Handler eine V-Operation auf dem Semaphor aus

Der Geräteprozess wird entblockiert (und dadurch rechenwillig) und kann das Gerät bedienen

Notizen

Anwendung: Durchsetzen einer Vorrangrelation

Hochschule RheinMain

Sei $\mathbb{P}$ eine Menge von kooperierenden Prozessen (ein *Prozesssystem*), und sei „<“ eine partielle Ordnung auf der Menge $\mathbb{P}$ mit:

$$P_1 < P_2 :\Leftrightarrow \text{Prozess } P_1 \text{ muss vor } P_2 \text{ ausgeführt werden;}$$

„<“ wird **Vorrangrelation** genannt.

Graphisch dargestellt als **gerichteter azyklischer Graph**[4], wird dabei häufig $P_1 < P_2$ dargestellt durch: $P_1 \rightarrow P_2$

Jeder Vorrangbeziehung $P_1 < P_2$ wird ein Semaphor `s` zugeordnet, auf das P_1 `V(s)` und P_2 `P(s)` ausführt.

Alle Semaphore werden mit 0 initialisiert.

→ Dann können alle Prozesse gleichzeitig gestartet werden. Die Vorrangrelation zwischen ihnen wird korrekt durchgesetzt.

[4] *directed acyclic graph* - DAG

Notizen

Beispiel: Vorrangrelation

Hochschule RheinMain

```
P1() {
.. work ..
 V(S_a);
 V(S_b);
 exit();
}
```

```
P2() {
 P(S_a);
.. work ..
 V(S_c);
 exit();
}
```

```
P3() {
 P(S_b);
.. work ..
 V(S_d);
 exit();
}
```

```
P4() {
 P(S_c);
 P(S_d);
.. work ..
 exit();
}
```


Notizen

P() und V() in Betriebssystemen

Hochschule RheinMain

Die meisten (alle?) Betriebssysteme bieten Dienste zum Umgang mit Semaphoren

In der Regel heißen diese aber *nicht* P() und V()

Beispiele:

- POSIX Threads Interface (pthreads): Mutex-Funktionen `phtread_mutex_xxx()` implementieren binäre Semaphoren (vgl. 5.2.2)
- UNIX System V Release 4 (SVR4): Funktionen `semget()`, `semop()`, `semctl()` arbeiten auf *Mengen* von semaphoren
- Windows NT: Zählsemaphore als Systemobjekte, Funktionen: `CreateSemaphore()`, `ReleaseSemaphore()`, `OpenSemaphore()`, `WaitForSingleObject()`

Notizen

Pthread Mutex Beispiel (1)

Hochschule RheinMain

Ziel: Alle Threads führen ihren „kritischen Abschnitt“ (in rot) jeweils nacheinaner ohne Überschneidungen aus.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread(void *arg)
{ /* hier: arg ist Zeiger auf integer (s.u.) */
    int i, num = *(int*)arg;
    pthread_mutex_lock(&mutex);   /* prolog */
    for(i = 0; i < 5; i++)
    {
        printf("Thread %d mit i=%d\n", num, i);
        sleep(1);
    }
    pthread_mutex_unlock(&mutex); /* epilog */
    pthread_exit((void*)0);
}
```


Notizen

Pthread Mutex Beispiel (2)

Hochschule RheinMain

Notizen

`main()`-Programm dazu...

```
#define THREADS 5
int main()
{
    void *rc;
    pthread_t threads[THREADS];
    int arg[THREADS];
    for(int i = 0; i < THREADS; i++)
    {   /* erzeuge 5 Threads */
        arg[i] = i; /* Thread-Nummer als Argument uebergeben */
        if(!!pthread_create(&threads[i], NULL, thread, &arg[i]))
        {
            fprintf (stderr, "Konnte Thread %d nicht starten\n", i);
            exit(EXIT_FAILURE);
        }
    }
    for(int i = 0; i < THREADS; i++)
    {   /* Auf Ende von 5 Threads warten */
        pthread_join(threads[i], &rc);
    }
    return(EXIT_SUCCESS);
}
```


Höhere Synchronisationsprimitive

Hochschule RheinMain

Notizen

Condition Variable

Synchronisationsobjekt des Betriebssystems.

Warteschlange für wartende Threads.

Wechselseitiger Ausschluss durch „Support-Mutex“

Mutex wird beim Warten implizit freigegeben[5]:

```
cond_wait(cond_t *cv, mutex_t *m) {
    mutex_unlock(m);
    warte_im_betriebssystem(cv, ...);
    mutex_lock(m);
}
```

[5] Nur zur Verdeutlichung: **Kein** echter Programmcode

POSIX Condition Variables

Hochschule RheinMain

Synchronisation von Threads basierend auf Bedingung über aktuellem Wert einer Variable

Benutzung immer im Zusammenhang mit assoziiertem Mutex

Condition Variable

- deklarieren vom Typ `pthread_cond_t`
- `pthread_cond_init()` (Initialisieren)
- `pthread_cond_destroy()` (Zerstören)

Warten und Signalisieren

- `pthread_cond_wait(condition,mutex)` Blockieren bis Bedingung signalisiert wird
- `pthread_cond_signal(condition)` Signalisieren der Bedingung (Wecken mindestens eines anderen Threads)
- `pthread_cond_broadcast(condition)` Signalisieren der Bedingung (Wecken aller wartenden Threads)

Details siehe Manpages oder:
`https://computing.llnl.gov/tutorials/pthreads/`

Notizen

POSIX Condition Variables

Hochschule RheinMain

Typische Verwendung

```
/* gemeinsame Daten */
pthread_mutex_t m;
pthread_cond_t cv;
Bool tuwas = FALSE;
```

```
// Wartender Thread
pthread_mutex_lock(&m);
...
while (!tuwas) {
    pthread_cond_wait(&cv, &m);
}
...
pthread_mutex_unlock(&m);
```

```
// Signalisierender Thread
pthread_mutex_lock(&m);
...
tuwas = TRUE;
pthread_cond_signal(&cv);
...
...
pthread_mutex_unlock(&m);
```


Notizen

Monitore

Hochschule RheinMain

Vorschlag Hoare (1974), Brinch Hansen (1975):

In Programmiersprache eingebettete Primitive zur einfacheren Entwicklung korrekter nebenläufiger Programme.

Ein **Monitor** besteht aus

- Variablen und Datenstrukturen (die das eigentliche Betriebsmittel darstellen)
- Sichtbaren Operationen zur Manipulation der Daten.
- Wechselseitiger Ausschluss in der Ausführung der Operationen ist garantiert, d.h. zu jedem Zeitpunkt ist höchstens ein Prozess im Monitor aktiv.

Code zur Sicherstellung des wechselseitigen Ausschlusses wird durch den Compiler erzeugt, z.B. auf der Basis von Semaphoren.

Heutige Sicht:

- Monitor entspricht einer Instanz eines abstrakten Datentyps mit automatischem wechselseitigen Ausschluss
- Vergleiche Java „synchronized instance method“

Notizen

Monitore (2)

Hochschule RheinMain

Programmierung von Synchronisationsvorgängen innerhalb von Monitoren mit interner Condition Variablen mit Operationen `wait()` und `signal()`.

`wait(cv)` blockiert den Aufrufer, ein evtl. wartender anderer Aufrufer einer Monitor-Operation kann nun den Monitor betreten.

`signal(cv)` weckt einen Prozess (aus der Sicht des Monitors zufällig aus der Menge der Wartenden ausgewählt), der Aufrufer muss den Monitor sofort verlassen (Annahme: letzte Anweisung, Brinch Hansen-Modell). Falls kein Prozess wartet, ist `signal(cv)` ohne Wirkung (keine Zähler-Semantik!).

Monitore haben kaum Eingang in Programmiersprachen gefunden (Gegenbeispiele: Mesa (Xerox), eingeschränkt Java synchronized)).

Notizen

Monitore in Java

Hochschule RheinMain

Keyword: `synchronized`

macht aus einer Methode einen Monitor

→ impliziter kritischer Bereich

`wait()`

- blockiert den aufrufenden Thread
- Monitor ist danach wieder frei

`notify()`

- weckt einen beliebigen Thread

`notifyAll()`

- weckt alle wartenden Threads

Notizen

Beispiel: Erzeuger/Verbraucher-Problem

Hochschule RheinMain

monitor: hypothetische C-Erweiterung

Lösung mit Monitor

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    count++;
    if (count == 1) signal(empty);
  }

  int remove(void) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Der Verbraucher muß warten, solange der Puffer leer ist

Ein wegen der empty- Bedingung blockierter Prozess, könnte nun weitermachen

Ein wegen der full-Bedingung blockierter Prozess, könnte nun weitermachen

Notizen

Weitere Ansätze

Hochschule RheinMain

Weitere, hier nicht besprochene Synchronisationsprimitive:

- Eventcounts: Reed, Kanodia (1979)
- Serializer: Atkinson, Hewitt (1979)
- Objekte mit Pfadausdrücken (path expressions) zur Festlegung von zulässigen Ausführungsfolgen der Operationen einschl. deren Nebenläufigkeit:
- Read/Write-Locks (vgl. Datenbanken).
- Barrieren
- fork/join

Literatur hierzu: z.B. Tanenbaum, Peterson/Silberschatz, Maekawa et. al.

Notizen

Klassische Synchronisationsprobleme

Hochschule RheinMain

Gliederung

- Erzeuger-Verbraucher-Problem
- Philosophen-Problem
- Leser-Schreiber-Problem

Notizen

Erzeuger-Verbraucher-Problem

Probleme

Erzeuger: Will Einfügen, aber Puffer ist voll.
Lösung: Lege dich schlafen, lass dich vom Verbraucher wecken, wenn er ein Datum entnommen hat.

Verbraucher: Will Entnehmen, aber Puffer ist leer.
Lösung: Lege dich schlafen, lass dich vom Erzeuger wecken, wenn er ein Datum eingefügt hat.

Notizen

Erzeuger-Verbraucher-Problem (2)

Problem wird auch „Problem des beschränkten Puffers“ (*bounded buffer problem*) genannt.

Sieht einfach aus, enthält aber kritische Abläufe.

Mögliche Erweiterung: mehrere Erzeuger, mehrere Verbraucher.

Betrachtete Lösungsansätze:

Lösung mit Semaphoren

Lösung mit Monitoren

Lösung mit Nachrichtenaustausch

Notizen

Lösung mit Semaphoren (1)

Hochschule RheinMain

Notizen

Wurde bereits besprochen (vgl. 5.2.3)

```
#define N 100           /* Kapazitaet des Puffers       */

/* gemeinsame Variablen
*/
semaphore mutex = 1;    /* kontrolliert krit. Bereich   */
semaphore empty = N;    /* zaehlt leere Eintraege       */
semaphore full  = 0;    /* zaehlt belegte Eintraege     */

void erzeuger(void)     /* Erzeuger                     */
{
    int item;
    while (TRUE) {
        produce_item(&item);  /* erzeuge Eintrag              */
        P(&empty);            /* besorge freien Platz         */
        P(&mutex);            /* tritt in krit. Abschnitt ein */
        enter_item(item);     /* fuege Eintrag in Puffer ein  */
        V(&mutex);            /* verlasse krit. Bereich       */
        V(&full);             /* erhoehe Anz. belegter Eintr. */
    }
}
```


Lösung mit Semaphoren (2)

Hochschule RheinMain

Notizen

```
void verbraucher(void)/* Verbraucher                  */
{
    int item;
    while(TRUE) {
        P(&full);           /* belegter Eintrag vorhanden?   */
        P(&mutex);          /* tritt in krit. Abschnitt ein  */
        remove_item(&item); /* entnimm Eintrag aus Puffer    */
        V(&mutex);          /* verlasse krit. Bereich        */
        V(&empty);          /* erhoehe Anz. freier Eintraege */
        consume_item(item); /* verarbeite Eintrag            */
    }
}
```

Semaphore `mutex` wird zur **Durchsetzung des wechselseitigen Ausschlusses** in der Benutzung des Puffers verwendet.
Semaphore `empty` und `full` werden zur **Synchronisation von Erzeuger und Verbraucher** benutzt, um bestimmte Operationsreihenfolgen zu erreichen bzw. zu vermeiden.
(Beispiel: Erzeuger: `P(&empty)`, Verbraucher: `V(&empty)` → Wenn Puffer voll:
⇒ Erzeuger blockiert, bis Verbraucher einen Eintrag entnimmt.)
Algorithmen gelten unverändert auch für mehrere Erzeuger und/oder Verbraucher.

Lösung mit Monitor (1)

Hochschule RheinMain

Wurde bereits besprochen (vgl. 5.2.4)

```
monitor ErzeugerVerbraucher {
  condition full, empty;
  int count = 0;

  void insert(int item) {
    if (count == MAX) wait(full);
    /* 'item' am Ende einfuegen */
    enter_item(item);
    count++;
    if (count == 1) signal(empty);
  }

  int remove(int *item_p) {
    if (count == 0) wait(empty);
    /* vorderstes 'item' entnehmen */
    remove_item(item_p);
    count--;
    if (count == MAX - 1) signal(full);
    return item;
  }
}
```

Erzeuger muß warten, solange der Puffer voll ist

Ein wegen der empty-Bedingung wartender Prozess, könnte nun weitermachen

Der Verbraucher muß warten, solange der Puffer leer ist

Ein wegen der full-Bedingung blockierter Prozess, könnte nun weitermachen

Notizen

Lösung mit Monitor (2)

Hochschule RheinMain

Beachte: Die Ausführung der Operationen `insert` und `remove` ist wechselseitig ausgeschlossen (Monitor!).

Die Variable `count` gibt die augenblickliche Anzahl der belegten Pufferplätze an.

`full` und `empty` sind Condition Variablen, **keine Zähler**.

Algorithmen gelten unverändert auch für mehrere Erzeuger und/oder Verbraucher.

Notizen

Lösung mit Nachrichtenaustausch (1)

Hochschule RheinMain

Grundlage: Funktionen zum Nachrichtenaustausch

`send(process, int* message)` - Nachricht an Prozess senden

`receive(process, int* message)` - Nachricht von Prozess empfangen

```
#define N      100              /* Kapazitaet des Puffers       */
#define MSIZE    4              /* Nachrichtengroesse           */

typedef int message[MSIZE];     /* Nachrichtentyp               */

void producer(void)             /* Erzeuger                     */
{
   int item;
   message m;

   while (TRUE) {
      produce_item(&item);      /* erzeuge Eintrag              */
      receive(consumer, &m);    /* warte auf leere Nachricht    */
      build_message(&m, item);  /* erzeuge zu sendende Nachricht*/
      send(consumer, &m);       /* sende Nachricht z Verbraucher*/
   }
}
```


Notizen

Lösung mit Nachrichtenaustausch (2)

Hochschule RheinMain

```
void consumer(void)             /* Verbraucher                  */
{
   int item, i;
   message m;

   for (i = 0; i < N; i++)      /* sende N leere Nachrichten    */
      send (producer, &m);

   while (TRUE) {
      receive(producer, &m);    /* empfange Nachricht v Erzeuger*/
      extract_item(&m, &item);  /* entnimm Eintrag              */
      send(producer, &m);       /* sende leere Nachricht zurueck*/
      consume_item(item);       /* verarbeite Eintrag           */
   }
}
```

Beachte: **Kein** gemeinsamer Speicher zw. Erzeuger und Verbraucher.
Annahmen: Nachrichten haben feste Länge. Gesendete, noch nicht empfangene Nachrichten werden automatisch gepuffert.
Insgesamt `N` Nachrichten-„Hülsen“ (Umschläge) werden benutzt. Leere Nachrichten werden vom Verbraucher an den Erzeuger gesendet, der Erzeuger füllt leere Nachrichten mit Daten und sendet „gefüllte Umschläge“ an den Verbraucher.
Wegen direkter Adressierung sind diese Algorithmen nur für **einen** Erzeuger und **einen** Verbraucher geeignet.

Notizen

Dining Philosophers Problem

Hochschule RheinMain

Dijkstra (1965): *dining philosophers problem*.

5 PhilosophInnen sitzen am runden Tisch. Jede(r) hat einen Teller, zwischen je zwei benachbarten Tellern liegt eine Gabel. Linke und rechte Gabel werden zum Essen benötigt. Nach dem Essen werden beide Gabeln abgelegt. Essen und Denken wechseln einander fortlaufend ab.

Wie lautet der Algorithmus für die PhilosophInnen?

Notizen

Eine zu einfache, **fehlerhafte** Lösung

Hochschule RheinMain

```
#define N 5              /* Anzahl der PhilosophInnen */

void philosopher(int i)      /* i:0..N-1, welche(r)? */
{
   while (TRUE) {
      think();               /* Denken               */
      take_fork(i);          /* Greife linke Gabel   */
      take_fork((i+1)%N);    /* Greife rechte Gabel  */
      eat();                 /* Essen                */
      put_fork(i);           /* Ablegen linke Gabel  */
      put_fork((i+1)%N);     /* Ablegen rechte Gabel */
   }
}
```

Gedankenexperiment: Was passiert wenn ...

... Alle die linke Gabel durch `take_fork(i)` nehmen,

... und dann die rechte Gabel durch `take_fork((i+1)%N)` zu nehmen versuchen?

→ Dann sind alle für immer blockiert!

→ Es liegt ein sogenannter **Deadlock** vor.

Notizen

Verbesserte, immer noch fehlerhafte Lösung

Hochschule RheinMain

Verbesserung: Nach dem Aufnehmen der linken Gabel prüfe, ob rechte Gabel verfügbar ist. Falls nicht, lege linke Gabel ab, warte eine Zeitlang, dann beginne von vorn.

```
#define N 5            /* Anzahl der PhilosophInnen */

void philosopher(int i)                /* i:0..N-1, welche(r)? */
{
    while (TRUE) {
        think();                        /* Denken               */
        do {
            take_fork(i);                /* Greife linke Gabel   */
            ok = try_take_fork((i+1)%N);  /* Versuche rechte Gabel*/
            if(!ok) {                    /* Erfolg?              */
                put_fork(i);               /* nein -> linke zurueck*/
                wait();                    /* warte                */
            }
        } while(!ok);
        eat();              /* Essen                */
        put_fork(i);        /* Ablegen linke Gabel  */
        put_fork((i+1)%N);  /* Ablegen rechte Gabel */
    }
}
```

Dieser Ansatz vermeidet den alten Fehler (Deadlock), enthält aber einen neuen:
Gedankenexperiment: Alle greifen gleichzeitig die linke Gabel, erkennen, dass die rechte nicht verfügbar ist, legen die linke wieder ab, warten gleich lang, greifen wieder gleichzeitig die linke usw.
Dieses Problem der endlosen Ausführung ohne Fortschritt wird „Verhungern“ (*Starvation*) genannt.

Notizen

Korrekte, aber **unbefriedigende** Lösung

Hochschule RheinMain

```
#define N 5            /* Anzahl der PhilosophInnen */
semaphore mutex = 1;

void philosopher(int i)      /* i:0..N-1, welche(r)? */
{
    while (TRUE) {
        think();                /* Denken                */
        P(&mutex);              /* Beginn krit. Bereich */
        take_fork(i);           /* Greife linke Gabel    */
        take_fork((i+1)%N);     /* Greife rechte Gabel   */
        eat();                  /* Essen                 */
        put_fork(i);            /* Ablegen linke Gabel   */
        put_fork((i+1)%N);      /* Ablegen rechte Gabel */
        V(&mutex);              /* Ende krit. Bereich    */
    }
}
```

Semaphore schützt gesamten „Ess-Abschnitt“

Kein Deadlock, aber unbefriedigend:
nur ein(e) PhilosophIn kann gleichzeitig essen !

Notizen

Lösung mit Semaphoren (1)

Hochschule RheinMain

```
#define N           5        /* Anzahl der Philosophen              */
#define LEFT  (i-1+N)%N      /* Nummer des linken Nachbarn von i    */
#define RIGHT (i+1)%N        /* Nummer des rechten Nachbarn von i   */
#define THINKING    0        /* Zustand: Denkend                    */
#define HUNGRY      1        /* Zust: Versucht, Gabeln zu bekommen */
#define EATING      2        /* Zustand: Essend                     */

/* gemeinsame Variablen                                            */
int state[N];                /* Zustaende aller PhilosophInnen      */
semaphore mutex = 1;         /* fuer wechselseitigen Aussschluss    */
semaphore s[n];              /* Semaphor fuer jeden Philosoph       */

void philosopher(int i) {  /* i:0..N-1, welcher Philosoph     */
    while (TRUE) {
        think();             /* Denken                              */
        take_forks(i);       /* Greife beide Gabeln oder blockiere */
        eat();               /* Essen                               */
        put_forks(i);        /* Ablegen beider Gabeln               */
    }
}
```


Notizen

Lösung mit Semaphoren (1)

Hochschule RheinMain

```
void take_forks(int i) /* i:0..N-1, welche(r) PhilosophIn?  */
{
    P(&mutex);           /* tritt in krit. Bereich ein          */
    state[i] = HUNGRY;   /* zeige, dass du hungrig bist         */
    test(i);             /* versuche, beide Gabeln zu bekommen */
    V(&mutex);           /* verlasse krit. Bereich              */
    P(&s[i]);            /* bockiere, falls Gabeln nicht frei   */
}

void put_forks(int i) /* i:0..N-1, welche(r) PhilosophIn?   */
{
    P(&mutex);           /* tritt in krit. Bereich ein          */
    state[i] = THINKING;/* zeige, dass du fertig bist          */
    test(LEFT);          /* kann linker Nachbar jetzt essen ?   */
    test(RIGHT);         /* kann rechter Nachbar jetzt essen ? */
    V(&mutex);           /* verlasse krit. Bereich              */
}

void test(int i) /* i:0..N-1, welche(r) PhilosophIn? */
{
    if (state[i]==HUNGRY &&
        state[LEFT]!=EATING && state[RIGHT]!=EATING) {
        state[i]=EATING;/* jetzt kann Phil i essen !           */
        V(&s[i]);        /* "sage es ihm"                       */
    }
}
```


Notizen

Bemerkungen

Hochschule RheinMain

Jeder Prozess führt die Prozedur `philosopher` als Hauptprogramm aus, die anderen Prozeduren sind gewöhnliche Unterprogramme, keine separaten Prozesse.

Das Array `state` speichert die aktuellen Zustände der Philosophen: essend, denkend, hungernd (versucht, Gabeln zu bekommen).

Jeder Prozess blockiert an einem ihm zugeordneten Semaphor `s[i]`, wenn die benötigten Gabeln nicht verfügbar sind.

Das Semaphor `mutex` sichert den kritischen Abschnitt der Benutzung der Zustandsinformation.

Die Lösung ist korrekt, sie enthält keinen Deadlock und kein Verhungern.

Die Lösung lässt eine möglichst hohe Nebenläufigkeit zwischen den Philosophen zu.

Die Lösung ist für eine beliebige Anzahl von PhilosophInnen korrekt.

Notizen

Leser-Schreiber-Problem

Hochschule RheinMain

Zu jedem Zeitpunkt dürfen entweder mehrere Leser oder ein Schreiber zugreifen.

Verboten: gleichzeitiges Lesen und Schreiben

Wie sollten Leser- und Schreiber-Programme aussehen?

Notizen

Lösung mit Semaphoren (1)

Hochschule RheinMain

Lesezugriff:

```
/* gemeinsame Variablen: */
semaphore mutex = 1;      /* wechsels. Aussschluss fuer rc      */
semaphore db = 1;         /* Semaphor fuer Datenbestand         */
int rc = 0;               /* readcount: Anzahl Leser            */

void reader(void)         /* Leser                              */
{
   while (TRUE) {
      P(&mutex);          /* erhalten exkl. Zugriff auf rc      */
      rc = rc + 1;        /* ein zusaetzlicher Leser            */
      if (rc==1)          /* Erster Leser?                      */
          P(&db);         /* ja -> reserviere Daten             */
      V(&mutex);          /* freigeben exkl. Zugriff auf rc     */

      read_data_base();   /* lies Datenbestand                  */

      P(&mutex);          /* erhalten exkl. Zugriff auf rc      */
      rc = rc - 1;        /* ein Leser weniger                  */
      if (rc==0)          /* letzter Leser ?                    */
          V(&db);         /* ja -> Daten freigeb.               */
      V(&mutex);          /* freigeben exkl. Zugriff auf rc     */
      use_data_read();    /* unkrit. Bereich                    */
   }
}
```


Notizen

Lösung mit Semaphoren (2)

Hochschule RheinMain

Schreibzugriff:

```
void writer(void)         /* Schreiber                          */
{
   while (TRUE) {
      create_data();      /* unkrit. Bereich                    */
      P(&db);             /* erhalten exkl. Zugriff auf Daten   */
      write_data_base();  /* schreibe Datenbestand              */
      V(&db);             /* freigeben exkl. Zugriff auf Daten  */
   }
}
```

`mutex` sichert krit. Abschnitt bezüglich des Read-Counters `rc`.
`db` sichert Zugriff auf den Datenbestand, so dass **entweder** mehrere Leser **oder** ein Schreiber zugreifen können.
Der erste Leser führt eine P-Operation auf `db` aus, alle weiteren inkrementieren nur `rc`.
Der letzte Leser führt eine V-Operation auf `db` aus, so dass ein wartender Schreiber Zugriff erhält.
Die **Lösung bevorzugt Leser**: Neu eintreffende Leser erhalten Zugriff vor einem schon wartenden Schreiber, wenn noch mindestens ein Leser Zugriff hat.

Notizen

Zusammenfassung (1)

Hochschule RheinMain

Was haben wir in Kap. 5 gemacht?

Interaktionen zwischen Prozessen können zu zeitkritischen Abläufen führen, d.h. Situationen, in denen das Ergebnis vom zeitlichen Ablauf abhängt. Zeitkritische Abläufe führen zu einem nicht reproduzierbaren Verhalten und müssen vermieden werden.

Kritische Bereiche als Teile von Programmen, in denen mit anderen Prozessen gemeinsamer Zustand manipuliert wird, bieten die Möglichkeit des wechselseitigen Ausschlusses. Sie vermeiden damit zeitkritische Abläufe und erlauben komplexe unteilbare (atomare) Aktionen.

Notizen

Zusammenfassung (2)

Hochschule RheinMain

Viele Primitive zur Synchronisation und Kommunikation von Prozessen wurden vorgeschlagen. Sie machen verschiedene Annahmen über die unterlagerten elementaren unteilbaren Operationen, sind aber im Prinzip gleich mächtig. Besprochen wurden insbesondere Semaphoren, Monitore, Condition Variablen und Nachrichtenaustausch, die in aktuellen Systemen weit verbreitet sind.

Es gibt eine Reihe von klassischen Problemen der Interprozesskommunikation, an denen die Nutzbarkeit neuer vorgeschlagener Primitive gezeigt wird. Von diesen wurden das Erzeuger-Verbraucher-Problem, das Philosophen-Problem und das Leser-Schreiber-Problem besprochen.

Auch mit den heute üblichen Primitiven muss sorgfältig umgegangen werden, um inkorrekte Lösungen, Deadlocks und Starvation zu vermeiden.

Notizen

6. Prozesskommunikation

https://www.animierte-gifs.net/data/media/1798/animiertes-richter-bild-0001.gif

Notizen

Prozesskommunikation

Ereigniskommunikation
Nachrichtenkommunikation
Zusammenfassung

Notizen

Ereigniskommunikation

Hochschule RheinMain

Bisher:

Prozesse kooperieren durch gemeinsame Benutzung von Betriebsmitteln (z.B. Speicherbereichen) (Sharing-Paradigma)
Synchronisationsmechanismen zur Vermeidung von zeitkritischen Abläufen
Synchronisationsmechanismen auch zur Signalisierung von Ereignissen zwischen Prozessen geeignet (z.B. Fertig-Signale zur Durchsetzung einer Vorrangrelation zwischen Prozessen eines Prozesssystems)

Hier:

Betriebssysteme können darüberhinaus spezielle Mechanismen für die Kommunikation (Signalisierung) von Ereignissen anbieten. (Diese Mechanismen können wiederum auch zur Synchronisation eingesetzt werden).
Kommunikation von Ereignissen ist ein Spezialfall der Nachrichten-orienterten Kommunikation (vgl. 6.2).

Notizen

Ereignisse

Hochschule RheinMain

Ereignisse zeigen das Eintreten einer „wichtigen Begebenheit“ an, z.B.:

- das Erreichen eines bestimmten Zustands
- das Eintreten eines Fehlers
- den Ablauf einer vorgegebenen Zeitspanne.

Ereigniskonzept als Übertragung des Hardware-Interrupt-Konzepts auf die Software-Ebene
Ereignisse werden lediglich nummeriert, Zuordnung einer tatsächlichen Bedeutung kann ganz oder teilweise vorgegeben oder den Anwendungen überlassen sein.
Bereitstellung eines effizienten Ereigniskonzepts ist insbesondere für Echtzeitbetriebssysteme von Bedeutung:
In Echtzeitanwendungen modellieren Ereignisse häufig reale Ereignisse in der Außenwelt.

Notizen

Signale in UNIX (1)

Hochschule RheinMain

Eigenschaften:

UNIX unterscheidet zwei Arten von Signalen:

- (Synchrone) Fehler-Ereignisse, die bei der Programmausführung auftreten, z.B.:
  - Adressierung eines ungültigen Speicherbereichs (→ SIGSEGV)
  - Versuch der Ausführung einer ungültigen Instruktion (→ SIGILL)
- Asynchrone Ereignisse, die von außerhalb des Prozesses stammen, z.B.:
  - Beendigung eines Kind-Prozesses (→ SIGCHLD)
  - Terminalleitungstreiber zeigt Ende der Übertragung an (→ SIGHUP)
  - Programmierter Timer läuft ab (→ SIGALRM)

Das UNIX-Signalkonzept überträgt des Hardware-Interrupt-Konzepts auf die Software-Ebene:

Interrupt	$\cong$	Signal
Interrupt Handler	$\cong$	Signal Handler
Maskieren von Interrupts	$\cong$	Maskieren von Signalen

Notizen

Signale in UNIX (2)

Hochschule RheinMain

UNIX unterscheidet eine feste Anzahl von Signalen

- Linux: `SIG_MAXSIG` = 64
- SysV: `MAXSIG` = 32

i.d.R. definiert durch Wortlänge → Signalmenge darstellbar als Bitmaske
Die Bezeichnung der Signale und ihr zugehöriger Zahlenwert sind z.T. für verschiedene UNIX Versionen (Linux, BSD, SysV) unterschiedlich.
Typische Signale (Linux, komplette Liste mit: `kill -l`):

Signal	Wert	Bedeutung	Default-Aktion
SIGHUP	1	Hangup, Verbindungsabbruch	exit
SIGILL	4	ungültiger Maschinenbefehl	core dump
SIGABRT	6	Abbruch des Prozesses mit Dump	core dump
SIGFPE	8	Gleitkommafehler	core dump
SIGKILL	9	Kill-Signal, **nicht abfangbar**	exit
SIGUSR1	10	Anwender-definierbar 1	exit
SIGSEGV	11	ungültige Speicheradresse	core dump
SIGUSR2	12	Anwender-definierbar 2	exit
SIGPIPE	13	Schreiben in Pipe ohne Empfänger	exit (vgl. 6.2)
SIGALRM	14	Ablauf des Weckers	exit
SIGTERM	15	Aufforderung zur Terminierung	exit
SIGCHLD	17	Statusänderung Kindprozess	ignore
SIGCONT	18	Fortsetzung	ignore
SIGTSTP	20	Anwender-Stop von tty	stop

Notizen

Signale in UNIX (3)

Ein Prozess im Zustand **rechnend** kann durch den BS-Kern Signale **senden und empfangen**.
Nicht-rechnende Prozesse können nur Signale von rechnenden Prozessen **empfangen**.
Ein Prozess kann festlegen, dass er bestimmte Signale selbst durch einen **Signal-Handler** behandeln oder **ignorieren** will (s.u.). Einige Signale (wie z.B. `SIGKILL`) können jedoch nicht abgefangen werden.
Ein Prozess kann nur Signale an Prozesse mit **derselben** realen oder effektiven **User-Id** senden (Prozessgruppe), insbesondere an sich selbst.
Darüber hinaus können Prozesse mit der realen oder effektiven **User-Id 0** (d.h. mit Superuser-Berechtigung) Signale an **beliebige Prozesse** senden.
Der BS-Kern kann jedes Signal an jeden Prozess senden.
Die Verarbeitung von anstehenden Signalen geschieht, wenn der Prozess aktiv ist und im Kernmodus ausgeführt wird.
Prozesse können Signale **maskieren**. Die Signalzustellung wird dann vorübergehend ausgesetzt.
Das Betriebssystem hält **Signalmasken** im Prozesskontrollblock des Prozesses (vgl. 2.1).

Notizen

Systemdienste zur Signalisierung (1)

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Sende das Signal `sig` an den Prozess `pid` (Mitglied derselben Prozessgruppe) oder an alle Prozesse dieser Gruppe, falls `pid` Null ist. Weitere Möglichkeiten für Prozesse mit User-Id 0 (Superuser).

```
int sigpause(int sig);
```

Das Signal `sig` wird aus der Signalmaske entfernt (d.h. zugelassen), und der Prozess blockiert, bis ein Signal empfangen oder der Prozess beendet wird.

```
int pause(void);
```

Der Prozess blockiert, bis ein Signal empfangen oder der Prozess beendet wird.

```
int sighold(int sig);
```

Das Signal `sig` wird der Signalmaske hinzugefügt und dadurch bis auf weiteres zurückgehalten (maskiert).

Notizen

Beispiel: Programmstart mit Timeout-Abbruch

Hochschule RheinMain

```
#include <...>
#define TIMEOUT 5
int main(int argc, char *argv[]) {
    int pid, status;
    if(pid=fork()) {
        fprintf(stderr,"Kind pid=%d gestartet\n",pid);
        sleep(TIMEOUT);
        if(waitpid(pid,&status,WNOHANG)==0) {
            fprintf(stderr,"Timeout, Abbruch!\n");
            kill(pid, SIGKILL);
            wait(&status);
        }
        fprintf(stderr,"Kind endet, Status=%d\n",status);
    } else {
        execve(argv[1], argv+1, NULL);
        fprintf(stderr,"Fehler beim Starten von %s",argv[1]);
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

```
$ ./timeoutdemo /bin/sleep 6
Kind pid=28407 gestartet
Timeout, Abbruch!

Kind endet, Status=9
```

Kind beendet sich nach weniger als TIMEOUT Sekunden: OK
Kind läuft länger als TIMEOUT Sekunden ⇒ Abbruch mit `kill()`

Notizen

Systemdienste zur Signalisierung (2)

Hochschule RheinMain

```
int sigaction(int sig,
struct sigaction *act, struct sigaction *oldact);
```

Zuordnung einer Aktion bei Auftreten des Signals `sig` und Speichern der bisher gültigen Aktion gemäß POSIX. Erfolgt i.d.R. nur einmal im Programm für jedes Signal, für das die Default-Aktion geändert werden soll.
Dabei ist:

```
struct sigaction {
   int sa_flags;          /* Flags, POSIX: nur SA_NOCLDSTOP */
   void (*sa_handler)();  /* Adresse einer eigenen Funktion */
                          /* oder SIG_IGN oder SIG_DFL      */
   sigset_t sa_mask;      /* zu maskierende Signale, wenn   */
}                         /*    handler ausgefuehrt wird.   */
```

Fehler `EINVAL`, falls `sig` = `SIGKILL` oder `SIGSTOP`.
Im Falle von `sig==SIGCHLD` bewirkt `SA_NOCLDSTOP`, dass `SIGCHLD` erst gesendet wird, wenn alle Kindprozesse beendet sind (Normalfall).
Nach Abschluss der erfolgreichen Behandlung erfolgt die Fortsetzung des Programms an der unterbrochenen Stelle.

Notizen

Beispiel: Signal abfangen

```c
#include <...>

void myIntHandler(int sig) {
    fprintf(stderr,"Autsch!\n");
}

int main(void) {
    int i;
    struct sigaction act, old;
    act.sa_handler = myIntHandler;
    sigaction(SIGINT, &act, &old);
    for (i=0; i < 10; i++) {
        printf("Runde %d\n",i);
        sleep(1);
    }
    sigaction(SIGINT, &old, NULL);
    return 0;
}
```

```
$ ./sigactionhandler
Runde 0
Runde 1
Runde 2
^CAutsch!
Runde 3
Runde 4
Runde 5
^CAutsch!
Runde 6
^CAutsch!
Runde 7
Runde 8
Runde 9
```

SIGINT (Ctrl-C) wird abgefangen!

Notizen

Systemdienste zur Signalisierung (3)

`int sigrelse(int sig);`

Das Signal `sig` wird aus der Signalmaske entfernt und damit wieder zugelassen.

`int sigignore(int sig);`

Das Signal `sig` wird auf die Disposition `SIG_IGN` gesetzt und damit bis auf weiteres vom aufrufenden Prozess ignoriert.

`int alarm(int seconds);`

Das Signal `SIGALRM` wird nach Ablauf der übergebenen Zeitdauer zugestellt („Wecker stellen“).

wird im Praktikum vertieft.

Notizen

Nachrichtenorientierte Kommunikation

Hochschule RheinMain

Motivation:

Signalisierung von Ereignissen ist Spezialfall des Austausches von Nachrichten beliebiger Art
(Paradigma der Kooperation durch Nachrichtenaustausch).

Betriebssysteme besitzen i.d.R. entsprechende Mechanismen zur nachrichtenorientierten Kommunikation.

Nachrichtenorientierte Kommunikation ist sowohl für lokale wie auch für verteilte Rechensysteme anwendbar.

Hier:

Besprechung der Grundlagen eines solchen allgemeinen Nachrichten(austausch)systems zur Interprozesskommunikation.

Notizen

Nachrichtensystem (Message System)

Hochschule RheinMain

Aufgabe

Mechanismus, so dass Prozesse einander Nachrichten zukommen lassen können, ohne über gemeinsamen Speicher verfügen zu müssen.

Operationen

`SEND ( message, ... )` zum Senden einer Nachricht durch einen **Sendeprozess** oder einfach **Sender**.
`RECEIVE ( message, ... )` zum Empfangen einer Nachricht durch einen **Empfangsprozess** oder **Empfänger**.

Kopplung/Verbindung der Kommunikationspartner

notwendig, um die Kommunikation zu ermöglichen.
Verbindung heißt **Kommunikationskanal** oder **Kanal**.
Ein Prozess kann zu einem Zeitpunkt mehrere Kanäle aufrecht erhalten (i.d.R. zu verschiedenen Prozessen).

Wichtige Entwurfsaspekte eines Nachrichtensystems

Adressierung, Pufferung, Nachrichtenstruktur

Notizen

Kommunikationskanal

Hochschule RheinMain

Anzahl der Kommunikationsteilnehmer eines Kanals

Genau zwei: Regelfall.
Mehr als zwei: Gruppenkommunikation (Multicast-Dienst, Spezialfall: Broadcast, d.h. Rundsendung an alle), hier nicht weiter betrachtet.

Richtung des Nachrichtenflusses eines Kanals

Kanal heißt **gerichtet** oder **unidirektional**, wenn ein Prozess ausschließlich die Sender-Rolle, der andere ausschließlich die Empfänger-Rolle ausübt, ansonsten **ungerichtet** oder **bidirektional**.

Notizen

Adressierung (1)

Hochschule RheinMain

Direkte Adressierung

Prozesse haben eindeutige Adressen.
Benennung ist **explizit** und **symmetrisch**:
sendender Prozess muss Empfänger benennen und umgekehrt.

- `SEND ( P, message )`
  Sende eine Nachricht an Prozess `P`.
- `RECEIVE ( Q, message )`
  Empfange eine Nachricht von Prozess `Q`.

Asymmetrische Variante (z.B. für Server-Prozesse):

- Sender benennt Empfänger, Empfänger (Server-Prozess) wird mit dem Empfang die Identität des Senders bekannt:
- `SEND ( P, message )`
- `RECEIVE ( sender_id , message )`

Notizen

Adressierung (2)

Hochschule RheinMain

Indirekte Adressierung

Kommunikation erfolgt indirekt über zwischengeschaltete **Mailboxes** (Puffer für eine Anzahl von Nachrichten):

- `SEND ( mbox, message )`
  Sende eine Nachricht an Mailbox `mbox`.
- `RECEIVE ( mbox, message )`
  Empfange eine Nachricht von Mailbox `mbox`.

Vorteile:

- Verbesserte Modularität.
- Prozessmenge kann transparent restrukturiert werden, z.B. nach Ausfall eines Empfangsprozesses.
- Erweiterte Zuordnungsmöglichkeiten von Sendern und Empfängern, wie z.B. m:1, 1:n, m:n.

Beispiel: m:1

Notizen

Pufferung

Hochschule RheinMain

Definition: Kapazität

Die **Kapazität eines Kanals** bezeichnet die Anzahl der Nachrichten, die vorübergehend in einem Kanal gespeichert werden können, um Sender und Empfänger zeitlich zu entkoppeln.

Die Pufferungsfähigkeit eines Kanals wird i.d.R. durch einen Warteraum / Warteschlange im Betriebssystemkern erreicht.

Möglichkeiten:

Keine Pufferung (Kapazität Null)
Beschänkte Kapazität
Unbeschänkte Kapazität

Notizen

Keine Pufferung (Kapazität Null)

Hochschule RheinMain

Der Sender wird blockiert, wenn die SEND-Operation vor der entsprechenden RECEIVE-Operation stattfindet. Wird dann die entsprechende RECEIVE-Operation ausgeführt, wird die Nachricht ohne Zwischenspeicherung **unmittelbar** vom Sender- zum Empfänger-Prozess kopiert.
Findet umgekehrt RECEIVE zuerst statt, so wird der Empfänger bis zum Aufruf der SEND-Operation blockiert.
Diese ungepufferte Kommunikationsform, die Sender und Empfänger zeitlich sehr eng koppelt, heisst **Rendezvous** oder **synchroner Nachrichtenaustausch**.
Beispiele:

- Hoare: Communicating Sequential Processes (CSP).
- Kommunikation zwischen Ada-Tasks.
- Mikrokern „L4“ [J. Liedtke 1995. *On μ-Kernel construction*].

Synchroner Nachrichtenaustausch wird häufig als zu inflexibel angesehen.

Notizen

Beschränkte Kapazität

Hochschule RheinMain

Der Kanal kann zu einem Zeitpunkt maximal N Nachrichten enthalten (Warteraum der Kapazität N).
Im Falle einer SEND-Operation bei nicht-vollem Warteraum wird die Nachricht im Warteraum abgelegt, und der Sendeprozess fährt fort.
Ist der Warteraum voll (enthält N gesendete aber noch nicht empfangene Nachrichten), so wird der Sender blockiert, bis ein freier Warteplatz vorhanden ist.
Das Ablegen der Nachricht kann durch Kopieren der Nachricht oder Speichern eines Zeigers auf die Nachricht realisiert sein.
Analog wird der Empfänger bei Ausführung einer RECEIVE-Operation blockiert, wenn der Warteraum leer ist.
Vgl. 5.3.1 (Erzeuger-Verbraucher-Problem):

Notizen

Unbeschränkte Kapazität

Hochschule RheinMain

Der Kanal kann potenziell eine unbeschränkte Anzahl von Nachrichten enthalten.

→ Die SEND-Operation **kann nicht blockieren**.

Lediglich der Empfänger kann bei Ausführung einer RECEIVE-Operation blockieren, wenn der Warteraum leer ist.

Implementierung ist möglich, indem Sender und Empfänger die Warteplätze beim Aufruf „mitbringen“.

Notizen

Pufferung: Konsequenzen

Hochschule RheinMain

Gepufferte Kommunikation (beschränkt oder unbeschränkt) bewirkt zeitlich lose Kopplung der Kommunikationspartner.
Nach Abschluss der SEND-Operation weiß der Sender nicht, ob der Empfänger die Nachricht erhalten hat. Er kennt i.d.R. auch keine maximale Zeitdauer dafür.
Wenn dieses Wissen wesentlich für den Sender ist, muss dazu eine explizite Kommunikation zwischen Sender und Empfänger durchgeführt werden:

Prozess P (Sender):		Prozess Q (Empfänger):
…		…
`SEND( Q, message);`	⟶	`RECEIVE( P, message);`
`RECEIVE( Q, reply);`	⟵	`SEND( P, "acknowledge");`
…		…

Notizen

Semantik von Nachrichten (1)

Hochschule RheinMain

Typisierte Nachrichten

Nachrichten haben eine typisierte Struktur.

Typ ist Sender und Empfänger und z.T. dem Nachrichtensystem bekannt und wird in Operationen verwendet.

Beispielhafter Aufbau einer Nachricht:

Notizen

Semantik von Nachrichten (2)

Hochschule RheinMain

Nachrichtencontainer

Nachrichten sind für Sender und Empfänger identifizierbare Einheiten fester oder variabler Länge. Nachrichtengrenzen bleiben erhalten.

Korrekte Interpretation der internen Struktur einer Nachricht obliegt den Kommunikationspartnern.

Beispiel: UNIX message queues.

Bytestrom

Empfänger (und das Nachrichtensystem) sehen ausschließlich eine Folge von Zeichen (Bytestrom).

Übergebene Nachrichten verschiedener SEND-Operationen sind als Einheiten nicht mehr identifizierbar. Nachrichtengrenzen gehen verloren.

Beispiel: UNIX pipes.

Notizen

Implementierung

Hochschule RheinMain

Klassische Implementierung

Speicher für zu puffernde Nachrichten liegt im BS-Kern.
Bei Ausführung von `SEND` wird die Nachricht aus dem sendenden Prozess(-adressraum) in den BS-Kern kopiert.
Bei Ausführung von `RECEIVE` wird die Nachricht aus dem BS-Kern in den empfangenden Prozess(-adressraum) kopiert.

→ 2-maliges Kopieren. Aufwändig!

Integration mit Speicherverwaltung

In modernen Betriebssystemen wird die Implementierung von Nachrichtenkommunikation mit dem Virtual Memory Management integriert (siehe Kap. 8, z.B. Mach).
Versenden von Nachrichten wird auf das Kopieren von Deskriptoren von Speicherbereichen beschränkt.
Die Speicherbereiche werden durch Markierung als *copy-on-write* nur im Notfall wirklich kopiert (lazy copying, vgl. Kap. 8).

Notizen

Beispiel: UNIX (1)

Hochschule RheinMain

UNIX stellt mehrere Dienste zum Nachrichtenaustausch zwischen Prozessen zur Verfügung:

(Anonyme) Pipes
- Ursprünglicher Mechanismus zum **uni**direktionalen Nachrichtentransport (Bytestrom) zwischen verwandten Prozessen
- Vererbung der Kommunikationsendpunkte durch fork().

Named Pipes oder FIFOs
- Erweiterung auf nicht-verwandte Prozesse.
- Dateisystem-Namensraum als gemeinsamer Namensraum zur Benennung von Named Pipes.
- Repräsentierung von Named Pipes im Dateisystem.

Message Queues
- Bestandteil der System V IPC-Mechanismen.
- Message Queue ist komplexes, gemeinsam benutztes Objekt zum Austausch typisierter Nachrichten zwischen potentiell beliebig vielen Sende- und Empfangs-Prozessen.

Notizen

Beispiel: UNIX (2)

Hochschule RheinMain

Sockets

- In 4.2BSD UNIX zusammen mit TCP/IP eingeführtes Konzept zur allgemeinen, rechnerübergreifenden, bidirektionalen, nachrichtenorientierten Interprozesskommunikation.
- zur Programmierung von Client/Server-Kommunikation geeignet.
- weit verbreitet (wird in LV „Verteilte Systeme“ behandelt).

STREAMS

- Mit UNIX System V Rel. 3 erstmals eingeführte Gesamtumgebung zur Entwicklung von geschichteten IPC-Protokollen.
- Modularisierung durch verkettbare STREAMS-Module.
- In UNIX System V Rel. 4 werden Pipes, Named Pipes, Terminal-Protokolle und Netzwerkkommunikation in der STREAMS-Umgebung bereitgestellt.

Notizen

Beispiel: Anonyme Pipes

Hochschule RheinMain

Eigenschaften:

Anonyme Pipe definiert namenlosen unidirektionalen Kommunikationskanal, über den zwei Prozesse mit einem gemeinsamen Vorfahren einen Bytestrom kommunizieren können.

Pipe besitzt eine Pufferkapazität von einer Seite (z.B. 4 Kb)

wird von den Prozessen wie eine Datei behandelt.

Typischer Umgang:

Elternprozess legt Pipe an, erzeugt zwei Kindprozesse durch `fork()`, die die geöffneten Enden der Pipe als File-Deskriptoren erben.

Der Schreiber schließt (mittels `close()`) die Lese-Seite, der Leser die Schreib-Seite, der Elternprozess beide Seiten.

Analog kann der Elternprozess mit einem Kindprozess über eine Pipe kommunizieren.

Notizen

Anonyme Pipes: Systemdienste

Hochschule RheinMain

`int pipe(int fds[2])`

Erzeugen einer anonymen geöffneten Pipe. Der File-Deskriptor `fds[0]` kann zum Lesen, `fds[1]` zum Schreiben benutzt werden.

`int write(int fd, char* buf, unsigned length)`

Senden von `length` bytes in die durch `fd` bezeichnete Pipe. Aufrufer blockiert, falls Pipe voll ist. Nicht-blockierendes Schreiben ist nach `fcntl()` mit `O_NDELAY` möglich. Write erzeugt `SIGPIPE`-Signal, falls Leser-Seite geschlossen wurde.

`int read(int fd, char* buf, unsigned length)`

Empfangen von maximal `length` Bytes aus der durch `fd` bezeichneten Pipe. Aufrufer blockiert, falls Pipe leer ist. (Blockierung kann analog zu `write()` vermieden werden). Rückgabewert ist die tatsächlich gelesene Anzahl Bytes, 0 bei Schließen der Pipe durch die Senderseite (Dateiende), -1 bei Fehler.

`int close(int fd)`

Schließen eines Endes der Pipe zum Beenden der Kommunikation.

Notizen

Beispiel: Eltern↔Kind Kommunikation

Hochschule RheinMain

`pipe(int fds[2])` erzeugt **zwei** File-Deskriptoren:

- `fds[0]` ist zum Lesen geöffnet
- `fds[1]` zum Schreiben

Umkehrung: Kind→Eltern Kommunikation
Fehlerbehandlung (hier weggelassen):
Rückgabewert: 0 für OK, -1 für Fehler

Notizen

Beispiel: Pipes in der Shell

Hochschule RheinMain

Wie oft ist „hansl“ auf dem Rechner angemeldet?
→ Shell-Kommandozeile:

```
who | grep "hansl" | wc -l
```

Dazu erzeugt die Shell 2 Pipes und 3 Kindprozesse, deren Standardein-/-ausgabe-File-Deskriptoren sie wie folgt setzt:

Notizen

Beispiel: popen()

Hochschule RheinMain

```
#include <stdio.h>
#include <stdlib.h>
#define MAXZEILE 80
int main(void) {
  char zeile[MAXZEILE];
  FILE *fp;
  if ( (fp = popen("/bin/date", "r")) == NULL) {
    perror("Fehler bei popen"); exit(1);
  }
  if (fgets(zeile, MAXZEILE, fp)== NULL) {
    perror("Fehler bei fgets"); exit(2);
  }
  pclose(fp);
  printf("Ausgabe von 'date' ist: %s\n", zeile);
  return 0;
}
```

⇒ Ausgabe: `Ausgabe von 'date' ist: Di 5. Jan 14:10:47 CET 2021`
`popen()` startet ein Kommando als Subprozess, in dessen Standardeingabe geschrieben (“w“) **oder** dessen Standardausgabe gelesen (“r“) werden kann (entweder/oder)
Pipe ist dabei als „Stream“ (Typ `FILE*`) repräsentiert
→ Verwendung mit Stream-Funktionen (`fprintf()`, `fgets()`, ...)
Schließen mit `pclose()`

Notizen

Named Pipes oder FIFOs

Hochschule RheinMain

Notizen

Eigenschaften

Eine benannte Pipe, auch FIFO genannt, definiert einen Kommunikationskanal, über den mehrere Sender- und mehrere Empfänger-Prozesse, die nicht miteinander verwandt sein müssen, einen gemeinsamen Bytestrom kommunizieren können.

Eine benannte Pipe besitzt einen Namen aus dem Dateinamensraum und eine Repräsentierung im Dateisystem (*inode*, Dateikontrollblock, vgl. Kap. 10) aber keine Datenblöcke, sondern lediglich einen Seiten-Puffer im Arbeitsspeicher.

Für benannte Pipes existieren Zugriffsrechte wie bei Dateien, die beim Öffnen überprüft werden.

In System V sind FIFOs durch ein spezielles *vnode*-Dateisystem `fifofs` implementiert (vgl. Kap. 10).

Named Pipes: Systemdienste

Hochschule RheinMain

Notizen

`int mknod(char* pfad, int modus, int dev)`

Mit `mknod()` können allgemein beliebige Knoten im Dateisystem angelegt werden. Zum Erzeugen einer Named Pipe gibt `pfad` den (Datei-)Namen für das FIFO an, `modus` wird gebildet durch `S_IFIFO|<rechte>`, wobei die Bitmaske `<rechte>` die Lese-/Schreibrechte für user/group/others wie üblich kodiert (z.B. dürfen mit `0666` beliebige Prozesse lesen und schreiben), `dev` hat für FIFOs den Wert 0.
(In SysV.4 existiert zusätzlich `mkfifo()`).

`int open(char* pfad, int modus)`

Öffnen der Pipe mit Namen `pfad` zum Lesen (modus `O_RDONLY`) oder Schreiben (modus `O_WRONLY`) wie bei üblicher Datei. Öffnen blockiert, bis sowohl ein Leser als auch ein Schreiber vorhanden sind. Durch Setzen des Flags `O_NDELAY` wird nicht-blockierendes Lesen oder Schreiben ermöglicht.

`read()`, `write()` und `close()` wie bei anonymen Pipes.

Wird im Praktikum vertieft.

Beispiel: Named Pipes oder FIFOs

```
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void) {
    int fd;
    mknod("myfifo",S_IFIFO|0666,0);
    fd = open("myfifo", O_WRONLY);
    write(fd, ...);
    ...
    /* Schliessen des FIFO */
    unlink("myfifo");
}
```

```
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(void) {
    int fd;
    ...
    fd = open("myfifo", O_RDONLY);
 read(fd, ...);
    ...
}

(Fehlerbehandlung weggelassen)
```

`mknod()` erzeugt FIFO mit angegebenem Pfad / Zugriffsbits
Named Pipe (FIFO) erscheint wie eine Datei im Dateibaum
Kann daher von beliebigen Prozessen (nicht nur Parent/Child) auf dem Rechner „gesehen“ und mit den bekannten Dateioperationen genutzt werden (Zugriffsrechte vorausgesetzt)
Schließen über Datei-Löschoperation `unlink()` (!)

Notizen

Message Queues

Eine Message Queue ist eine **verkettete** (Nachrichten-) **Liste** (Nachrichtenwarteschlange), die vom Betriebssystemkern verwaltet wird.
Empfangsreihenfolge normalerweise „first-in-first-out“, eine Priorisierung der Nachrichten ist aber auch möglich
Im Rahmen der UNIX System V IPC-Mechanismen eingeführt (Bedeutung gesunken).
Wie alle System V IPC-Mechanismen sind Message Queues sind persistente, systemweite Objekte.
Shell Kommandos:

- `ipcmk -Q` → Message Queue anlegen
- `ipcrm -q <id>` → Message Queue löschen
- `ipcs -q` → Message Queue Objekte anzeigen

Notizen

Message Queues: Eigenschaften

Hochschule RheinMain

Eine Message Queue definiert einen **Kommunikationskanal**, über den **mehrere** Sender- und mehrere Empfänger-Prozesse, die **nicht miteinander verwandt** sein müssen, **typisierte** Nachrichten **variabler Länge** austauschen können.
Message Queues besitzen einen eindeutigen Bezeichner aus dem „Key-Namensraum".
Eine geöffnete Message Queue wird intern über einen Message Queue Identifier (Integer) bezeichnet.
Eine Nachricht besteht aus einem **Integer-Nachrichtentyp** und **variabel langem Nachrichteninhalt**. Festlegung der Bedeutung von Typ und Inhalt ist den Prozessen vorbehalten.
Eine Message Queue hält Berechtigungen zur Zugriffskontrolle.
Anzahl und Puffergröße der Message Queues werden bei der Systemgenerierung festgelegt.

Notizen

Message Queues: Systemdienste

Hochschule RheinMain

`int msgget(key_t key, int flag)`

Erzeugen einer neuen oder Öffnen einer existierenden Message Queue, liefert den internen Identifier der Message Queue.

```
int msgsnd(int id, struct msgbuf * buf, int msgsz,
int msgflag)
```

Senden einer Nachricht an die angegebene Message Queue.

```
int msgrcv(int id, struct msgbuf * buf, int msgsz,
long msgtype, int msgflag)
```

Empfangen einer Nachricht eines wählbaren oder beliebigen Typs von der angegebenen Message Queue. Der Aufrufer blockiert, falls keine Nachricht des gewünschten Typs vorhanden ist. Die Blockierung kann über ein Flag vermieden werden.

`int msgctl(int id, int cmd, struct msqid_ds * buf)`

Ausführung einer Kontroll-Operation auf der Message Queue zum Auslesen von Verwaltungs-Informationen, Verändern der Berechtigungen sowie zum Zerstören der Message Queue.

Notizen

Zusammenfassung

Hochschule RheinMain

Was haben wir in Kap. 6 gemacht?

Prozesse müssen kommunizieren können. Die grundlegenden Kommunikationsmöglichkeiten basieren auf der gemeinsamen Benutzung von Speicher (Sharing) sowie dem **Nachrichtenaustausch**.

Kommunikation von Ereignissen (Signalisierung) wurde am Beispiel UNIX erläutert.

Grundlagen der nachrichtenrichtenorientierten Kommunikation wurden vorgestellt und am Beispiel der UNIX Pipes, Named Pipes und Message Queues verdeutlicht.

Notizen

Notizen

7. Deadlocks

https://i.redd.it/rniab4wus2dz.jpg

Notizen

Deadlocks

Einführung
Betriebsmittel
Deadlocks
Verfahren zur Deadlock-Behandlung
Verwandte Fragestellungen
Zusammenfassung

Notizen

Einführung

Hochschule RheinMain

Motivation

In diesem Kapitel wird ein grundlegendes Problem bei der Benutzung exklusiv benutzbarer Betriebsmittel behandelt, das sogenannte **Deadlock-Problem**.

Ein Deadlock wird auch als **Systemverklemmungszustand** bezeichnet.

Untersuchungen zu Deadlocks nehmen breiten Raum in der frühen Betriebssystem-Literatur ein.

Ziel hier: Kennenlernen von Methoden zur Erkennung, Behebung, Vermeidung und Verhinderung von Deadlocks.

Notizen

Hinführendes Beispiel

Hochschule RheinMain

Beide Prozesse sind blockiert und bleiben es für immer.

Eine solche Situation heißt **Deadlock** (genauere Definition folgt).

Deadlocks treten auch in vielen anderen Situationen auf, z.B. beim Sperren von Datensätzen in Datenbanken.

Notizen

Betriebsmittel

Hochschule RheinMain

Reservierbare Objekte (Objekte, auf die Zugriff erteilt werden kann) heißen **Betriebsmittel**.

Diese können **Hard**- oder **Softwarekomponenten** sein:

- CD-Brenner
- Prozessor
- Ein Datensatz
- Eine Verwaltungsstruktur des Betriebssystems
- ...

Ein Betriebsmittel(typ) kann in mehreren identischen Instanzen oder Einheiten vorliegen

Ein Betriebsmittel ist **unterbrechbar**, wenn es einem Prozess ohne nachteilige Auswirkungen entzogen werden kann, ansonsten heißt es **ununterbrechbar**.

Beispiele

- Arbeitsspeicher → unterbrechbar (Prozess aus-/einlagern)
- Drucker, DVD-Brenner: ununterbrechbar

Notizen

Benutzung von Betriebsmitteln

Hochschule RheinMain

Schritte zur Benutzung eines Betriebsmittels:

- Anfordern eines BMs
- Benutzen des BMs
- Freigeben des BMs

Falls BM angefordert wird aber nicht verfügbar ist, muss der anfordernde Prozess warten. Warten kann durch Blockieren oder durch Busy Waiting realisiert sein (vgl Kap 5).

Form einer BM-Anforderung ist in konkreten Systemen sehr unterschiedlich, u.U. auch BM-Typ-abhängig. Typisch: `open(bm)`.

Bewilligung einer Betriebsmittelanforderung heißt auch (Betriebsmittel-)**Zuteilung**. Prozess wird durch Zuteilung **Inhaber** des Betriebsmittels.

Notizen

Deadlocks

Hochschule RheinMain

Definition

Eine Menge von Prozessen befindet sich in einem **Deadlock-Zustand**, falls jeder Prozess der Menge auf ein Ereignis wartet, das nur ein anderer Prozess der Menge auslösen kann.

Man sagt auch:
Die Prozesse sind in einen „Deadlock verstrickt“.

Da alle Prozesse warten, kann keiner jemals ein Ereignis erzeugen, auf das einer der anderen wartet. Alle Prozesse warten also für immer.

Notizen

Voraussetzungen für Deadlocks

Hochschule RheinMain

Coffman (1971): Voraussetzungen für die Entstehung von Deadlocks:

Wechselseitiger Ausschluß:
Jedes Betriebsmittel ist entweder frei oder genau einem Prozess zugeteilt

Belegungs-/Anforderungs-Bedingung (*„Hold-and-wait“*):
Prozesse können zu bereits reservierten Betriebsmitteln noch weitere anfordern

Ununterbrechbarkeit:
Einmal einem Prozess zugeteilte Betriebsmittel können nicht wieder ohne dessen Zustimmung (Freigabe) entzogen werden.

Zyklisches Warten:
Es muss eine zyklische Kette von Prozessen geben, in der jeder Prozess auf ein Betriebsmittel wartet, das dem nächsten Prozess in der Kette gehört.

Alle vier Bedinungen gleichzeitig erfüllt ⇒ Ein Deadlock **ist möglich**
(falls eine nicht erfüllt ist ⇒ **kein Deadlock möglich**)

Notizen

Belegungs-Anforderungs-Graphen

Hochschule RheinMain

Graphische Darstellung der Beziehung von Prozessen zu Betriebsmitteln (Holt, 1972)

Es gibt zwei Knotentypen:

- Prozesse, repräsentiert durch Kreise:
- Betriebsmittel, repräsentiert durch Quadrate:

Pfeile:

- P belegt B
- P wartet auf B

Zyklus im Graphen → Deadlock

Notizen

Beispiel

Hochschule RheinMain

Gegeben:

- drei Prozesse A, B, C und
- drei Betriebsmittel R,S,T

Prozess A	**Prozess B**	**Prozess C**
- Anforderung R	- Anforderung S	- Anforderung T
- Anforderung S	- Anforderung T	- Anforderung R
- Freigabe R	- Freigabe S	- Freigabe T
- Freigabe S	- Freigabe T	- Freigabe R

Das Betriebssystem kann jeden (nicht blockierten) Prozess **jederzeit** ausführen

Sequentielle Ausführung von A, B, C wäre unproblematisch (dann aber auch keine Nebenläufigkeit)

Wie sieht es bei nebenläufiger Ausführung aus?

Notizen

Ausführung I

Notizen

Ausführung II

(B zunächst suspendiert)

Notizen

Verfahren zur Deadlock-Behandlung

Hochschule RheinMain

Mit Betriebsmittelzuteilungsgraphen (*„Belegungs/Anforderungs-Graphen“*) lassen sich Deadlocks erkennen (→Zyklus im Graph)
Wie weiter verfahren?

- **Ignorieren** („Vogel-Strauß-Verfahren“)
- Deadlocks **erkennen** und **beheben**
- **Verhinderung** durch Planung der Betriebsmittelzuordnung (*deadlock avoidance*)
- **Vermeidung** durch Nichterfüllung (mindestens) einer der vier Voraussetzungen für Deadlocks (*deadlock prevention*)

Diese Strategien werden im folgenden untersucht.

Notizen

Ignorieren des Problems

Hochschule RheinMain

http://clipart.coolclips.com/480/vectors/tf05038/CoolClips_anim0613.png

„Vogel-Strauß-Algorithmus“

Ausdruck optimistischer Lebenshaltung:

„Deadlocks kommen in der Praxis sowieso nie vor“

...warum also dann Aufwand in ihre Vermeidung stecken?

Beispiel:

- UNIX-System mit z.B. 100 Einträge großer Prozesstabelle
- 10 Programme versuchen gleichzeitig, je 12 Kindprozesse zu erzeugen
- Deadlock nach 90 erfolgreichen fork()-Aufrufen (wenn keiner der Prozesse aufgibt)

Ähnliche Beispiele sind mit anderen begrenzt großen Systemtabellen möglich (z.B. inode-Tabelle)

Notizen

...manchmal nicht so gut

http://clipart.coolclips.com/480/vectors/tf05038/CoolClips_anim0613.png

→

http://www.instore.si/newsarticle/newsarticle/Septembra-v-Aldiju-nojevo-meso

Notizen

Deadlock-Erkennung und Behebung

Engl.: ***deadlock detection and resolution / recovery***

Vorgehensweise: Das Auftreten von Deadlocks wird vom Betriebssystem nicht verhindert. Es wird versucht, Deadlocks zu erkennen und anschließend zu beheben.

Betrachtet werden im folgenden:

- Deadlock-Erkennung mit einem Betriebsmittel je Klasse (Einfacher Fall)
- Deadlock-Erkennung mit mehreren Betriebsmitteln je Klasse (Allgemeiner Fall)
- Verfahren zur Deadlock-Behebung

Notizen

Deadlocks erkennen (Einfacher Fall)

Hochschule RheinMain

Vereinfachende Annahme: **Ein Betriebsmittel** je Betriebsmitteltyp

Vorgehen:

- erzeuge Belegungs-/Anforderungs-Graph
- suche nach Zyklen
- falls ein Zyklus gefunden wurde: Deadlock beheben (s.u.)

Wann wird die Untersuchung durchgeführt?

- bei jeder Betriebsmittelanforderung?
- in **regelmäßigen** Zeitabständen?
- wenn **„Verdacht“** auf Deadlock besteht
  (z.B. Abfall der CPU-Auslastung unter eine Grenze)

Notizen

Beispiele: Sicher?

Hochschule RheinMain

4 Prozesse, ein Betriebsmitteltyp (10 Stück vorhanden)

verfügbar: 10

Proz.	hat	max.
A	0	6
B	0	5
C	0	4
D	0	7

sicher!

z.B. sequenzielle Ausführung von A, B, C, D in beliebiger Reihenfolge ist möglich.

verfügbar: 2

Proz.	hat	max.
A	1	6
B	1	5
C	2	4
D	4	7

sicher!

C ist ausführbar,(→ dann 4 verfügbar) dann D, B, A möglich.

verfügbar: 1

Proz.	hat	max.
A	1	6
B	2	5
C	2	4
D	4	7

unsicher!

Differenz $max - hat$ immer $> verfgbar$. Deadlock, sobald irgend ein Prozess auf sein Maximum zugeht

Notizen

Beispiel

A belegt R und fordert S an.

B fordert T an.

C fordert S an.

D belegt U und fordert S und T an.

E belegt T und fordert V an.

F belegt W und fordert S an.

G belegt V und fordert U an.

Notizen

Deadlocks erkennen

Erweiterung: Mehrere (E_i-viele) Betriebsmittel je Betriebsmitteltyp i (z.B. mehrere Drucker)

Prozesse $P_1, \ldots, P_n$

$$E = (E_1, E_2, \ldots, E_m)$$

Betriebsmittelvektor E: Gesamtzahl der BM je Typ i

$$A = (A_1, A_2, \ldots, A_m)$$

Verfügbarkeitsvektor A: Gesamtzahl der BM je Typ i

Belegungsmatrix C: Zeile j gibt BM-Belegung durch Prozess j an („Prozess j belegt C_{jk} Einheiten von BM k")

$$C = \begin{pmatrix} C_{11} & C_{12} \ldots & C_{1m} \\ C_{21} & C_{22} \ldots & C_{2m} \\ \ldots & \ldots\ldots & \ldots \\ C_{n1} & C_{n2} \ldots & C_{nm} \end{pmatrix}$$

Anforderungsmatrix R: Zeile j gibt BM-Belegung durch Prozess j an („Prozess j belegt R_{jk} Einheiten von BM k")

$$R = \begin{pmatrix} R_{11} & R_{12} \ldots & R_{1m} \\ R_{21} & R_{22} \ldots & R_{2m} \\ \ldots & \ldots\ldots & \ldots \\ R_{n1} & R_{n2} \ldots & R_{nm} \end{pmatrix}$$

Notizen

Erkennungsalgorithmus

Zu Beginn sind alle Prozesse aus P unmarkiert
(Markierung heißt, dass der Prozess in keinem DL steckt)

Suche einen Prozess, der ungehindert durchlaufen kann, also einen unmarkierten Prozess P_i, dessen Zeile in der Anforderungsmatrix-Zeile R_i (komponentenweise) kleiner oder gleich dem Verfügbarkeitsvektor A ist

Kein passendes P_i gefunden? Dann → **Ende**

Gefunden? Dann kann P_i durchlaufen und gibt danach seine belegten Betriebsmittel zurück: $A = A + C_i$, wird markiert und es geht beim nächsten unmarkierten Prozess weiter

Beim Ende des Verfahrens sind **alle unmarkierten** Prozesse an einem **Deadlock beteiligt**.

Notizen

Beispiel

Ausführbar ist zunächst nur P_3
Freigabe $C_3 = (0120)$
⇒ $A = (2100) + (0120)$
⇒ $A = (2220)$
Nun ausführbar: P_2
(benötigt $R_2 = (1010)$)
Freigabe $C_2 = (2001)$
⇒ $A = (4221)$
Schließlich auch P_1 ausführbar
⇒ $A = (4231)$
⇒ Alle Prozesse markiert,
kein Deadlock aufgetreten.

Notizen

Beheben von Deadlocks

Hochschule RheinMain

Wie kann man auf erkannte Deadlocks reagieren?

Prozessunterbrechung

- Betriebsmittel zeitweise entziehen, anderem Prozess bereitstellen und dann zurückgeben
- Kann je nach Betriebsmittel schwer oder nicht möglich sein

Teilweise Wiederholung (*rollback*)

- System sichert regelmäßig Prozesszustände (*checkpoints*)
- Dadurch ist Abbruch und späteres Wiederaufsetzen möglich
- Arbeit seit letztem Checkpoint geht beim Rücksetzen verloren und wird beim Neuaufsetzen wiederholt (ungünstig z.B. bei seit Checkpoint ausgedruckten Seiten)
- Beispiel: Transaktionsabbruch bei Datenbanken

Prozessabbruch

- Härteste, aber auch einfachste Maßnahme
- Nach Möglichkeit Prozesse auswählen, die relativ problemlos neu gestartet werden können (z.B. Compilierung)

Notizen

Verhindern von Deadlocks

Hochschule RheinMain

Bisher: Erkennung von Deadlocks, gegebenenfalls „drastische“ Maßnahmen zur Auflösung

Annahme bisher: Prozesse fordern alle Betriebsmittel „auf ein Mal“ an (vgl. 7.4.2).

In den meisten praktischen Fällen werden BM jedoch nacheinander angefordert

Das Betriebssystem muss dann dynamisch über die Zuteilung entscheiden

Notizen

Verhindern von Deadlocks

Hochschule RheinMain

Kann man **Deadlocks** durch „geschicktes“ Vorgehen bei der Betriebsmittelzuteilung **von vornherein verhindern**?

Welche Informationen müssen dazu vorab zur Verfügung stehen?

Im folgenden betrachtet

- Betriebsmittelpfade (Grafische Veranschaulichung)
- Sichere und unsichere Zustände
- Der vereinfachte Bankiersalgorithmus für eine BM-Klasse
- Der Bankiersalgorithmus für mehrere BM-Klassen

Notizen

Betriebsmittelpfade

Hochschule RheinMain

Notizen

(Un-)Sichere Zustände

Hochschule RheinMain

Definition

Ein Systemzustand ist **sicher**, wenn er

keinen Deadlock repräsentiert und

es eine geeignete Prozessausführungsreihenfolge gibt, bei der alle Anforderungen erfüllt werden

(die also **auch dann** nicht in einen Deadlock führt, wenn alle Prozesse gleich ihre max. Ressourcenanzahl anfordern)

Sonst heißt der Zustand **unsicher**.

Bei einem sicherem Zustand kann das System **garantieren**, dass alle Prozesse bis zum Ende durchlaufen können.
Bei unsicherem Zustand ist das nicht garantierbar (aber auch nicht ausgeschlossen!).

Beispiel: Ein Prozess gibt ein BM zu einem „glücklichen Zeitpunkt“ kurzzeitig frei, wodurch eine Deadlock-Situation „zufällig“ vermieden wird. (→ „Glück“ nicht vorhersehbar)

„Unsicher“ bedeutet also nicht „Deadlock unvermeidlich“.

Notizen

Beispiel

Hochschule RheinMain

3 Prozesse A,B,C; jeweils mit BM-Besitz und max. Bedarf

ein Betriebsmitteltyp, 10x vorhanden

Zustand (a) ist sicher (es gibt eine DL-freie Lösung)

(b2) ist **nicht** sicher (A und C brauchen je 5, frei sind nur 4)

Notizen

Bankier-Algorithmus (1 BM-Klasse)

Hochschule RheinMain

- Dijkstra (wer sonst? 1965):
- Ein **Bankier kennt** die **Kreditrahmen** seiner Kunden.
- Er geht davon aus, dass **nicht alle** Kunden **gleichzeitig** ihre Rahmen **voll** ausschöpfen werden.
- Daher hält er **weniger Bargeld** bereit als die **Summe** der Kreditrahmen.
- Gegebenenfalls **verzögert** er die **Zuteilung** eines Kredits, bis ein anderer Kunde zurückgezahlt hat.
- **Zuteilung** erfolgt **nur**, wenn sie **"sicher"** ist (also letztlich alle Kunden bis zu ihrem Kreditrahmen bedient werden können).

Bankier = Betriebssystem, Bargeld = Betriebsmitteltyp,
Kunden = Prozesse, Kredit = BM-Anforderung

Notizen

Bankier-Algorithmus (2)

Hochschule RheinMain

Prüfe bei jeder Anfrage, ob die Bewilligung in einen sicheren Zustand führt:

Prüfe dazu, ob ausreichend Betriebsmittel bereitstehen, um **mindestens einen** Prozess **vollständig** zufrieden zu stellen.

Davon ausgehend, dass dieser Prozess nach Durchlauf seine Betriebsmittel freigibt: führe **Test** mit dem Prozess aus, der dann am nächsten am Kreditrahmen ist

usw., **bis alle** Prozesse positiv getestet sind;

Falls **ja**, kann die aktuelle Anfrage **bewilligt** werden.

Sonst: Anforderung **verschieben** (warten)

Notizen

Verallgemeinerter Bankier-Algorithmus

Hochschule RheinMain

Mehrere Betriebsmittelklassen

Datenstrukturen wie bei „Deadlockerkennung“ (7.4.2)

Matrizen mit belegten / angeforderten Betriebsmitteln

Vektoren mit BM-Bestand, verfügbaren BM und belegten BM je Betriebsmitteltyp

- E Betriebsmittelvektor
- A Verfügbarkeitsvektor
- C Belegungsmatrix
- R Anforderungsmatrix

Notizen

Beispiel

Hochschule RheinMain

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix} \text{ vorhanden}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ zugewiesen}$$

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \end{pmatrix} \text{ verfügbar}$$

$$P = \begin{pmatrix} 5 & 3 & 2 & 2 \end{pmatrix} \text{ belegt}$$

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \text{ angefordert}$$

Sicher? Ja, Ausführungsfolge P_4, P_1, P_5, ...ist möglich:

$$P_4 \rightarrow A = \begin{pmatrix} 2 & 1 & 2 & 1 \end{pmatrix}$$

$$P_1 \rightarrow A = \begin{pmatrix} 5 & 1 & 3 & 2 \end{pmatrix}$$

$$P_5 \rightarrow A = \begin{pmatrix} 5 & 1 & 3 & 2 \end{pmatrix}$$

$$P_2 \rightarrow A = \begin{pmatrix} 5 & 2 & 3 & 2 \end{pmatrix}$$

$$P_3 \rightarrow A = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix}$$

Notizen

Beispiel

Hochschule RheinMain

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix} \text{ vorhanden}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ zugewiesen}$$

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix} \text{ verfügbar}$$

$$P = \begin{pmatrix} 5 & 3 & 3 & 2 \end{pmatrix} \text{ belegt}$$

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \text{ angefordert}$$

P2 fordere ein BM 3 an (rot)

Sicher? Ja, Ausführungsfolge P_4, P_1, P_5, P_2,P_3 möglich:

$$P_4 \rightarrow A = \begin{pmatrix} 2 & 1 & 1 & 1 \end{pmatrix}$$

$$P_1 \rightarrow A = \begin{pmatrix} 5 & 1 & 2 & 2 \end{pmatrix}$$

$$P_5 \rightarrow A = \begin{pmatrix} 5 & 1 & 2 & 2 \end{pmatrix}$$

$$P_2 \rightarrow A = \begin{pmatrix} 5 & 2 & 3 & 2 \end{pmatrix}$$

$$P_3 \rightarrow A = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix}$$

also erhält P_2 ein BM3

Notizen

Beispiel

Hochschule RheinMain

$$E = \begin{pmatrix} 6 & 3 & 4 & 2 \end{pmatrix} \text{ vorhanden}$$

$$C = \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \text{ zugewiesen}$$

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \text{ verfügbar}$$

$$P = \begin{pmatrix} 5 & 3 & 4 & 2 \end{pmatrix} \text{ belegt}$$

$$R = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{pmatrix} \text{ angefordert}$$

Nun fordere auch P_5 ein BM 3 an
→ dann würde
$A = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$

Sicher? *Nein!*
→ daher Anfrage von P_5 blockieren

Notizen

Ist der Bankier-Algorithmus praktikabel?

Hochschule RheinMain

In der Praxis gibt es mehrere Probleme beim Einsatz:

- Prozesse können „maximale Ressourcenanforderung“ selten im Voraus angeben
- Anzahl der Prozesse ändert sich ständig
- Ressourcen können verschwinden (z.B. durch Ausfall)

Notizen

Deadlock-Vermeidung

Hochschule RheinMain

Deadlock-Verhinderung ist wenig praktikabel ☹

Ansatz: **Vermeidung** mindestens einer der vier Deadlock-**Voraussetzungen** (vgl 7.2)

- Wechselseitiger Ausschluss
- Belegungs-/Anforderungsbedingung („Hold-and-Wait“, d.h. zu reservierten BM weitere anforderbar)
- Ununterbrechbarkeit (kein erzwungener BM-Entzug)
- zyklisches Warten

Notizen

1. Wechselseitiger Ausschluß?

Hochschule RheinMain

Falls es keine exklusive Zuteilung eines Betriebsmittels an einen Prozess gibt, gibt es auch keine Deadlocks.

Beispiel: Zugriff auf Drucker

Einführung eines **Spool-Systems**, das

- Druckaufträge von Prozessen (schnell) entgegennimmt
- ggf. zwischenspeichert
- und der Reihe nach auf dem Drucker ausgibt

Entkopplung zwischen (konkurrierenden) Prozessen und dem (langsamen) Betriebsmittel

Vermeidung einer exklusiven Zuteilung des Betriebsmittels „Drucker“

Notizen

2. Belegungs-/Anforderungsbedingung?

Hochschule RheinMain

Vermeiden, dass neue Betriebsmittel-Anforderungen zu bereits bestehenden hinzukommen.

„Preclaiming“: Alle Anforderungen zu Beginn der Ausführung stellen („alles oder nichts“)

Vorteil: Wenn Anforderungen erfüllt werden, kann der Prozess sicher bis zum Ende durchlaufen (er hat ja dann alles, was er braucht)

Nachteil:

- Anforderungen müssen **zu Beginn bekannt** sein
- Betriebsmittel werden unter Umständen **lange blockiert**
- und können zwischenzeitlich nicht (sinnvoll) anders genutzt werden.

Beispiel: Batch-Jobs bei Großrechnern.

Notizen

3. Ununterbrechbarkeit?

Hochschule RheinMain

Hängt vom Betriebsmittel ab, aber

„gewaltsamer“ Entzug ist in der Regel nicht akzeptabel

- Drucker?
- CD-Brenner?

Notizen

4. Zyklische Wartebedingung?

Hochschule RheinMain

Wenn es kein zyklisches Auf-einander-warten gibt, entstehen auch keine Deadlocks

Idee:

- Betriebsmitteltypen **linear ordnen** und
- nur in aufsteigender Ordnung Anforderungen annehmen (wenn mehrere Exemplare eines Typs gebraucht werden: alle Exemplare auf einmal anfordern)
- z.B. „Drucker *vor* Scanner *vor* CD-Brenner *vor* …“

Dadurch entsteht **automatisch** ein **zyklenfreier** Belegungs-Anforderungs-Graph,

wodurch Deadlocks ausgeschlossen sind.

Tatsächlich praktikables Verfahren.

Notizen

Deadlock-Vermeidung im Überblick

Hochschule RheinMain

Deadlock-Vermeidung durch Verhinderung (mindestens) einer der 4 Vorbedingungen eines Deadlocks ist möglich:

Wechselseitiger Ausschluß	→ Spooling
Belegungs-/Anforderungsbed.	→ Preclaiming
Ununterbrechbarkeit	(BM-Entzug...besser nicht)
Zyklisches Warten	→ Betriebsmittel ordnen

Notizen

Verwandte Fragestellungen

Hochschule RheinMain

Deadlocks bei der Benutzung von Semaphoren (vgl. Kap. 3)

Zwei-Phasen-Locking in Datenbanken

Verhungern (Starvation), kein Deadlock, aber auch kein Fortschritt für einen Prozess (vgl. Philosophen-Problem)

Notizen

Zusammenfassung (1)

Hochschule RheinMain

Deadlocks sind ein Problem in jedem Betriebssystem aber auch in nebenläufigen Anwendungssystemen.

Deadlocks treten auf, wenn einer Menge von Prozessen jeweils exklusiv ein Betriebsmittel zugeteilt ist, und alle ein Betriebsmittel anfordern, das bereits von einem anderen Prozess der Menge belegt ist. Alle Prozesse sind blockiert, keiner kann jemals fortgeführt werden.

Deadlocks können durch vier notwendige Bedingungen charakterisiert werden:

- Bedingung des wechselseitigen Ausschlusses
- Belegungs- / Anforderungs-Bedingung
- Ununterbrechbarkeitsbedingung
- Zyklische Wartebedingung

Notizen

Zusammenfassung (2)

Hochschule RheinMain

Deadlocks können vermieden werden (7.4.4), indem überprüft wird, ob der nachfolgende Zustand sicher ist oder nicht.

- Zustand ist sicher, wenn es Folge von Ereignissen gibt, so dass alle Prozesse ihre Ausführung beenden können.
- In unsicherem Zustand gibt es keine Garantie dafür.
- Der Bankier-Algorithmus vermeidet Deadlocks, in dem er Anforderungen zurückstellt, die das System in einen unsicheren Zustand überführen würden.

Deadlocks können durch konstruktive Verfahren verhindert werden (7.4.3)

- z.B. durch Vorabbelegen aller Betriebsmittel
- durch die geordnete Betriebsmittelbenutzung.

Notizen

Heute: Kleiner Ausflug in die Rechnerarchitektur

Kap. 8: Ca..

Cage ?

Cash ?

Cache !

© Robert Kaiser, Hochschule RheinMain

Wozu Caches? (1)

Hochschule RheinMain

- Prozessorgeschwindigkeit hat sich in den letzten 30 Jahren mehr als vertausendfacht
- Speicherzugriffsgeschwindigkeit ist zwar auch gestiegen, aber deutlich weniger:

Wozu Caches? (2)

Hochschule RheinMain

- Prozessortakt > 10 x Speichertakt ist heute keine Seltenheit → Speicher wird zum „Flaschenhals“
- Skalare Prozessoren: 1 Befehl pro Takt (*Super*skalare: noch mehr …)
- Wie soll das gehen?
- Ein Maschinenbefehl besteht mindestens aus
  - einem Speicherzugriff (Opcode fetch)
  - + evtl. auch noch mehrere Operanden
- Lösung: schneller, dafür kleiner Zwischenspeicher zw. Prozessor und Hauptspeicher: Cache(1)

(1) Engl. *Cache*: Depot, geheimes Versteck

Motivation

Hochschule RheinMain

- Cache gehört zur Mikroarchitektur, nicht zur ISA, ist also aus Programmierersicht transparent
- Warum sich hier damit beschäftigen?
- Gründe:
  - Erzielte/erzielbare Performance hängt von Lokalitätseigenschaften (s.u.) der Programme ab
    - Durch geschicktes/ungeschicktes Coden kann die Performance u.U. um Größenordnungen variieren
  - Im Bereich der Betriebssysteme sind Caches manchmal doch nicht so ganz transparent
    - DMA, Debuggen, Laden von Programmen
- → Grundlegende Kenntnisse über die Arbeitsweise von Caches sind erforderlich

Typisches Programmverhalten (Heuristik)

- **Räumliche Lokalität:** Zugriffe häufig auf Adresse in der Nähe bereits zuvor benutzer Adressen
- **Zeitliche Lokalität:** Zugriffe auf dieselbe oder benachbarte Adressen zeitlich nahe beieinander
- Trifft zu auf:
  - Befehlszugriffe: meistens (ausser bei Sprüngen)
  - Datenzugriffe: Programmabhängig
- Caches nutzen diese Eigenschaften:
  - Nachgefragte Daten und der sie umgebende Speicherblock (Cache-Zeile) werden möglichst lange zwischengespeichert
  - Wird das gleiche (oder ein benachbartes Datum) erneut nachgefragt, wird es aus dem Zwischenspeicher geliefert
- → geht wesentlich schneller

Begriff: Working Set

Hochschule RheinMain

- ***Working set***: Gesamtheit der Speicherobjekte, auf die ein Prozess während eines gegebenen Zeitintervalls zugreift
- **Ziel**: möglichst viel vom Working Set eines Prozesses im Cache halten
- Programme arbeiten i.d. R. auf verschiedenen „Sektionen“:
  - Programmcode (`.text`)
  - Initialisierte Daten (`.data`)
  - Uninitialisierte Daten (`.bss`)
  - Heap (z.B. von `malloc()`)
  - Stack
  - Evtl. shared Libraries

➔ Working Set besteht aus >5-6 „Regionen“

Begriffe: Hit und Miss

Hochschule RheinMain

- Programm greift auf ein Objekt (z.B. Variable) zu
- Zwei Möglichkeiten:

1. ***Cache Hit***: Daten sind bereits im Cache → schnell
2. ***Cache Miss***: Daten sind nicht im Cache, Hauptspeicherzugriff erforderlich → langsam

Begriffe: Konsistenz und Kohärenz

Hochschule RheinMain

- Der Cache hält **Kopien** von im Speicher liegenden Objekten → Gefahr von Inkonsistenzen
- Cache ist **konsistent** → Alle Cache-Kopien und Original im Hauptspeicher haben gleichen Wert
- Cache ist **kohärent** → Cache und Hauptspeicher liefern für das gleiche Objekt den gleichen Wert
- Kohärenz betrifft das korrekte Arbeiten von Programmen
  - Inkohärenz führt zu Fehlern → vermeiden!
  - (Vorübergehende) Inkonsistenz ist tolerabel, solange sie nicht zu Inkohärenz führt
- Es gibt verschiedene Verfahrensweisen um Kohärenz sicherzustellen (sog. Kohärenzprotokolle)
  → hier nur Grundlagen behandelt

Kohärenzprotokoll: Fallunterscheidung

Hochschule RheinMain

A) Bei **Lese**zugriff:

- ***Cache Hit*** → Daten aus dem Cache liefern, keine weiteren Aktionen
- ***Cache Miss*** → Daten werden aus dem Speicher geliefert, dabei in den Cache kopiert

B) Bei **Schreib**zugriff:

- ***Cache Hit*** → zwei Möglichkeiten:
  1) ***Write Trough***: Geschriebene Daten werden zugleich im Cache und im Speicher gespeichert (einfach aber langsam)
  2) ***Copy-Back***: Datum wird zunächst nur im Cache gespeichert, zugehörige Cache-Zeile als „dirty“ markiert
     → System wird vorübergehend inkonsistent
- ***Cache Miss*** → auch zwei Möglichkeiten:
  1) ***No Write Allocate***: Datum wird nur in Speicher geschrieben
  2) ***Write Allocate***: Datum wird in Speicher und Cache geschrieben (zusammen mit umliegender Cachezeile)

Cache-Effektivität (statistisch)

Hochschule RheinMain

- Effektive Wartezeit T_{eff} hängt ab von:
  - Wartezeit bei Zugriff auf Hauptspeicher (Cache miss): T_{miss}
  - Wartezeit bei Zugriff auf Cache (Cache hit): T_{hit}
  - Wahrscheinlichkeit eines Cache-Hit (*Trefferrate*): H

$$T_{eff} = H * T_{hit} + (1 - H) * T_{miss}$$

- Beispiel: H = 80 %, T_{hit} = 0, T_{miss} = 20ns:
  - => T_{eff} = 0,8 * 0 + 0,2 * 20ns = **4ns**

Mehrstufiger Cache (1)

Hochschule RheinMain

- z.B. Zweistufiger Cache (Level 1 und Level 2):

$$T_{eff} = H_{L1} * T_{L1hit} + (1 - H_{L1}) * (H_{L2} * T_{L2hit} + (1 - H_{L2}) * T_{miss})$$

- Beispiel: H_{L2} = 90 %, T_{L2hit} = 4ns, T_{miss} = 20ns:
  => T_{eff} = 0,8 * 0 + 0,2 * (0,9 * 4ns + 0,1 * 20ns) = **1,12ns**
- Caches heute meist mehrstufig:

Mehrstufiger Cache (2)

Hochschule RheinMain

- Heute bei PCs gängige Größen
  - L1 Cache: On-Chip, getrennt für Befehle und Daten, je 16-64kB
  - L2 Cache: On-Chip, gemeinsam für Befehle und Daten, 256kB bis 4MB
  - L3 Cache: Optional, Off-Chip (z.B. schnelles ECL-RAM mit 5ns Zugriffszeit)

Assoziativspeicher (1)

Hochschule RheinMain

- Bisher betrachtete Speicher ordnen Adressen → Daten zu
- Cache: hält Kopien relativ kleiner(1) Hauptspeicher-Ausschnitte
- Zuordnung zwischen Speicher-objekt und Cache-Zeile ergibt sich aus der Zugriffsreihenfolge
- **Kein** Zusammenhang zwischen Adresse eines Objekts und dem Ort seiner Cache-Kopie
- Adresse (bzw. Index) einer Cachezeile ist ohne Bedeutung
- Wie aber werden Cache-Zeilen adressiert?

(1) Cache-Zeilen Größe: einige zehn bis hundert Byte

Assoziativspeicher (2)

Hochschule RheinMain

- **Assoziativspeicher** (auch „inhaltsadressierter Speicher") (engl. *Content Adressable Memory* – CAM)
- **Assoziation von Wertepaaren**
  (hier: Speicheradresse und Cache-Zeile)
  (Auch der Mensch „speichert" assoziativ)
- Ein Cache-Eintrag besteht aus:
  - Einem Identifikationsteil, dem *Tag* (Etikett)
  - Einem Datenbereich (Cache-Zeile),
    der die eigentliche Kopie des Speicherinhaltes enthält
  - Einem Verwaltungsteil: z.B. Bits „V" (Valid) und „D" (Dirty)
    - V=1 → Cache-Zeile ist in Benutzung
    - D=1 → Inhalt der Cache-Zeile wurde verändert → ist inkonsistent

Assoziativspeicher (3)

Hochschule RheinMain

- Arbeitsweise:
  - Bei Zugriff auf ein Speicherobjekt: Adresse wird **gleichzeitig** mit den Tags **aller** Cache-Einträge verglichen
  - Existiert ein Eintrag mit übereinstimmendem Tag und V = 1 → *Cache Hit*: Gültige Kopie ist im zugehörigen Datenbereich
  - Existiert kein solcher Eintrag → *Cache Miss*

Assoziativspeicher (4)

Hochschule RheinMain

- Modellierung in C:

```
struct cache_eintrag {
  void *tag;
  int V, D;
  unsigned int cache_zeile[C
};

struct cache_eintrag cache[ANZ_CA

unsigned int *finde_eintrag(void *A   sse)
{
  for(i = 0; i < ANZ_CACHEZL; i++) {
    if(cache[i].tag == Adresse && cache[i].V)
      / * cache hit * /
      return(&cache[i].cache_zeile[0]);
  }
  return(NULL); / * cache miss * /
}
```

In Wirklichkeit: **parallele** Suche in Hardware

Verdrängungsstrategien

Hochschule RheinMain

- Wenn alle Cachezeilen belegt sind und eine Kopie eines Objekts in einer Zeile neu angelegt werden soll, so muss eine bestehende Cachezeile überschrieben („verdrängt“) werden
- Nach welcher Strategie erfolgt diese Auswahl?
- Hier besprochene Möglichkeiten:
  - **Random**: Ersetzung nach dem Zufallsprinzip
    - Einfach, überraschend gutes Ergebnis
  - **FIFO** (first in first out): Die am längsten im Cache befindliche Zeile wird ersetzt
    - Immer noch einfach, aber keine gute Idee
  - **LRU** (least recently used): Die am längsten nicht mehr verwendete Zeile wird ersetzt
    - Gute Ergebnisse, aber aufwändig
  - **LFU** (least frequently used): Die am wenigsten häufig verwendete Zeile wird ersetzt
    - Aufwand und Ergebnis ähnlich LRU (d.h. gut aber aufwändig)

Cache-Organisationsformen

Hochschule RheinMain

- Cache-Organisationsformen:
  1. Vollassoziativ: *fully associative*
  2. Direkt abbildend: *direct-mapped*
  3. Mehrfach assoziativ: *N-way set associative*
- Annahmen bei den folgenden Beispielen:
  - 32-bit Adressierung
  - 4K (4096) Byte Cache
  - Cache-Zeilengröße: 16 Byte
    (→ 4096 : 16 = 256 Cache-Einträge)

Vollassoziativer Cache: Aufbau

Vollassoziativer Cache: Beispiel

Vollassoziativer Cache

Hochschule RheinMain

- Ein Objekt kann in eine beliebige Cache-Zeile kopiert werden
  → Freie Auswahl eines freien / zu verdrängenden Eintrags
- Identifikation der Zeile ausschließlich anhand des Tags
  → Konsequenzen:
  - Tags **aller** Zeilen müssen mit Adresse verglichen werden
  - Vergleich muss **gleichzeitig** auf allen Zeilen erfolgen
  - Für jede Zeile wird ein eigener Vergleicher benötigt
  - Jedes Tag darf maximal ein Mal vorkommen
- Erreicht höchste Trefferquote (wg. Eintrags-Wahlfreiheit)
- Große Anzahl an Vergleichern (Im Beispiel: 256 für einen 4K Cache) → sehr hoher Hardwareaufwand
- Beispiel:
  - TLB-Cache des MIPS R3000 / R4000: Vollassoziativer Cache mit 64 / 128 Einträgen

Direkt abbildender Cache: Aufbau

Direkt abbildender Cache: Beispiel

Direkt abbildender Cache

Hochschule RheinMain

- Ein Teil der Adresse (im Beispiel: Bits 4 bis 11) wählt die Cachezeile aus
- Eindeutige Zuordnung ohne Wahlfreiheit, keine alternativen Verdrängungsstrategien möglich (aber auch keine nötig)
- Tag dient allein zur Hit/Miss Entscheidung
- **Konsequenzen**:
  - (+) Einfacher Aufbau (nur ein Vergleicher erforderlich)
  - (-) Schlechte Trefferquote
- **Beispiel**:
  - Ein Programm arbeitet in einer Schleife mit zwei Objekten, die in verschiedenen Cache-Zeilen liegen, deren Adressen sich aber in Bits 4 bis 11 nicht unterscheiden → Objekte verdrängen sich permanent gegenseitig (sog. „*Cache Trashing*“)

Mehrfach assoziativer Cache

Hochschule RheinMain

- Vollassoziativer Cache: Hohe Trefferquote, aber aufwändig
- Direkt abbildender Cache: Geringer Aufwand, aber schlechte Trefferquote (neigt zu „thrashing")
- Kompromiss: Mehrfach assoziativer Cache(*)
  - Zusammenfassen von je N (N = 2, 4, 8, ...) Cache-Zeilen zu einem „Satz" (engl. „*set*")
  - Ein Teil der Adresse dient als Satznummer
  - Innerhalb eines Satzes gibt es N mögliche Cache-Zeilen („Wege", engl. „*ways*"), die anhand ihres Tags unterschieden werden
- ➔ Für die Auswahl eines freien bzw. zu verdrängenden Cache-Eintrags stehen N Alternativen zur Verfügung
- Verdrängungsstrategien können –wenn auch eingeschränkt– umgesetzt werden

(*) engl. *N-way set associative cache*

z.B. zweifach assoziativer Cache: Aufbau

zweifach assoziativer Cache: Beispiel

Mehrfach assoziativer Cache: Fazit

Hochschule RheinMain

- Deutliche Verbesserung der Trefferquote gegenüber direkt abbildendem Cache
- N Wege → N Vergleicher werden benötigt
- Für N = 1 „degeneriert” er zum direkt abbildenden Cache
- Für N = <Anzahl der Cache-Zeilen> „degeneriert” er zum vollassoziativen Cache
- Für Zwischenwerte von N: guter Kompromiss zwischen Aufwand und Trefferquote
- Heute der am meisten verwendete Cache
- (s.o.) Working Set üblicher Programme besteht aus > 5-6 Regionen
- N sollte >= Anzahl der Regionen sein (sonst → Thrashing)

Mehrfach assoziativer Cache: Beispiel

Hochschule RheinMain

- Pentium 4:
  - L1 Datencache: 4-fach assoziativ (64 Byte Zeilengröße)
  - L1 Befehlscache: 8-fach assoziativ
  - L2 Cache: 8-fach assoziativ (64 Byte Zeilengröße)

Fallstricke

Hochschule RheinMain

- Potenzielle Probleme im Zusammenhang mit Caches
  - „zerklüfteter“ Working Set oder ungünstige Adresslage von Variablen kann zu *Thrashing* führen → drastischer Performance-Einbruch
  - **Multitasking**: Prozesswechsel bedeutet i.d.R. auch kompletten Wechsel des Working Set
  - ➔ Nach Prozesswechsel ist der Prozessor langsamer (u.U. bis Faktor 30!)
  - DMA und Schreibzugriffe auf Codespeicher: evtl. explizites *flush* & *invalidate* erforderlich (s.o.)
  - **Multicore:** Vielfach gemeinsamer L2/L3 Cache: → gegenseitiges „ausbremsen“ der Cores, wenn auf verschiedenen Working sets gearbeitet wird.

Schreib-Pufferspeicher (*Write Buffer*)

Hochschule RheinMain

- Charakteristisches Verhalten von –z.B.– C-Programmen:
  - Etwa 10% „store”-Befehle, d.h. speichern von Daten
  - Solche Schreibzugriffe kommen häufig in schneller Folge („Bursts”) vor (z.B. wenn zu Beginn eines Unterprogramms Register gerettet werden)
- Insbesondere bei einem *write through* Cache muss der Prozessor hier auf den langsamen Hauptspeicher warten
- Abhilfe durch *Write Buffer*:
  - Ausstehende Schreibzugriffe (Adressen **und** zu schreibende Daten) werden in einen FIFO-Puffer zwischengespeichert
  - ➔ Prozessor kann sofort weiterarbeiten
  - ➔ Zwischgespeicherte Speicherzugriffe werden parallel dazu abgearbeitet

Schreib-Pufferspeicher (*Write Buffer*)

Hochschule RheinMain

- Write Buffer finden sich z.B. bei ARM, PowerPC und MIPS-Prozessoren
- Potenzielles Problem: Lesezugriffe können Schreibzugriffe „überholen“
- z.B. bei Ein-/Ausgabe:
  - Gerät löst Interrupt aus, obwohl der bereits (per Schreibzugriff) abgeschaltet wurde
- Lösungswege:
  - **Software**: Puffer explizit „flushen“ (spezieller Maschinenbefehl)
  - **Hardware**: Jeder Lesezugriff wartet, bis der Puffer leer ist

9. Speicherverwaltung

https://www.pentagull.co.uk/news/esb-2101-released/

Notizen

Ausgangspunkt

Idealerweise sollte Speicher sein ...

- groß
- schnell
- nicht flüchtig[1]

In der Realität sind (heute) nicht alle Ziele gleichzeitig zu akzeptablen Preisen mit einem Speichermedium zu erreichen

Daher: Kombination verschiedener Speicherformen

[1] D.h.: „geht beim Ausschalten nicht verloren“

Notizen

Die Speicherhierarchie

Notizen

Worum geht es?

Definition: Speicherverwalter

Der **Speicherverwalter** ist der Teil des Betriebssystems, der den Arbeitsspeicher verwaltet.

Aufgaben

- Verwaltung von freien und belegten Speicherbereichen.
- Zuweisung von Speicherbereichen an Prozesse, wenn diese sie benötigen (Allokation).
- Freigabe nach Benutzung oder bei Prozessende.
- Durchführung von Auslagerungen von Prozessen (oder Teilen) zwischen Hauptspeicher und Sekundärspeicher bei Speicherengpässen.

In diesem Kapitel wird eine Folge von einfachen bis hochentwickelten Speicherverwaltungsverfahren betrachtet.

Notizen

Gliederung

Hochschule RheinMain

Speichermodell
Statische Speicherverwaltung
Swapping
Speicherverwaltungsalgorithmen
Virtueller Speicher
Seitenersetzungsalgorithmen
Entwurfsprobleme bei Paging-Systemen
Segmentierung
Beispiel: Speicherverwaltung in UNIX
Zusammenfassung

Notizen

Speichermodell

Hochschule RheinMain

Ein Programm und seine Daten werden während der Ausführung vollständig im Hauptspeicher gehalten[2]

Die Programm- und Datenobjekte lassen sich in verschiedene Klassen einteilen

Der Compiler/Linker ordnet die **statischen** (d.h. zur Compile-Zeit bekannten) Objekte je nach Klasse bestimmten „Sektionen“ zu:

Klasse	Lesen	Schreiben	Ausführen	Initialisiert	Sektion
Programmcode	x	-	x	x	`.text`
initialisierte Daten	x	x	-	x	`.data`
uninitialsierte Daten	x	x	-	-	`.bss`
nur-lese Daten	x	-	-	x	`.rodata`

Für jede Sektion wird ein entsprechend großes, zusammenhängendes Stück Speicher eingeteilt.

[2] Virtueller Speicher / Swapping / Paging vorerst nicht betrachtet

Notizen

Dynamische Daten

Hochschule RheinMain

Der restliche (freie) Speicher wird dynamisch, d.h. zur Laufzeit zugeteilt:

Heap: Dynamischer Speicher

- Wird in C mit `malloc()` zugeteilt und mit `free()` wieder freigegeben (In C++ mit `new` und `delete`)
- Wächst nach „oben“, d.h. zu zahlenmäßig höheren Adressen hin
- Anfang typischerweise direkt im Anschluss an statische Daten (`.bss`)

Stack: Lokale (dynamische) Variablen

- Sämtliche Automatic-Variablen in C (ausserdem alloca()-Funktion)
- Auch: Aufrufhierarchie (d.h. Rückkehradressen) und übergebene Parameter)
- Zuteilung / Freigabe automatisch durch Inkrementieren bzw. Dekrementieren des Stackpointers (z.B. mit `PUSH`- und `POP`-Befehlen)
- Wächst (i.d.R.) nach „unten“, d.h. zu niedrigeren Adressen hin
- Anfang typischerweise am oberen Ende des RAM-Speichers

→ **Stack und Heap wachsen einander entgegen. Wenn sie „zusammenstoßen“ geschehen „merkwürdige Dinge“ ...**

Notizen

Beispiel: Speicherlayout eines C-Programms

Hochschule RheinMain

ROM = Nur-Lese Speicher

RAM = Lese-/Schreib-Speicher

```
static int   a = 5;
char *s =    "Hallo";
int b;

main(int argc, char *argv[])
{
int i;
int l =      strlen(s);
char *p;
p =      malloc(l + 1);
    for(i = 0; i < l; i++)
       p[i]=  s[i];
}
```

`.text` und `.rodata` können von Prozessen gemeinsam benutzt werden, Daten, Stack und Heap sind privat.

Notizen

Das Relokationsproblem

Hochschule RheinMain

In der Regel sind Programme **nicht** Positionsunabhängig („PIC“=*position independent Code*, „PID“ = *position independent Data*)

Mögliche Lösung des Relokationsproblems: Loader addiert die Ladeadresse auf alle Adressen gemäß eines vom Binder erstellten Adressverzeichnisses des Programms.

Notizen

Anforderungen an Speicherverwaltung

Hochschule RheinMain

Schutz

Prozesse sollten nicht unerlaubt auf fremde Speicherbereiche zugreifen können

Gemeinsame Nutzung

Andererseits soll eine kontrollierte gemeinsame Nutzung von Speicherbereichen möglich sein
(z.B. 10 gleichzeitige bash-Nutzer → Code nicht 10x laden)

Relokation

Absolute Adressbezüge im Programmcode z.B. beim Laden in einen (anderen) konkreten Speicherbereich anpassen

Speicherorganisation

Unterstützung von Programm-Modularisierung durch Segmentierung, abgestufter Schutz z.B. für Daten / Code

Ein-/Auslagern von Speicherbereichen zwischen Hauptspeicher und Sekundärspeicher

Notizen

Statische Speicherverwaltung

Hochschule RheinMain

Definition: Statische Speicherverwaltung

Verfahren mit fester Zuordnung von Speicher an Prozesse.

Zuordnung ist keinen Veränderungen während der Laufzeit des Systems oder der Prozesse unterworfen.

(Nicht-statische Speicherverwaltung heißt auch dynamische Speicherverwaltung (ab Kap. 9.3).)

Im folgenden für den historischen Einprogramm- und Mehrprogrammbetrieb betrachtet.

Heute noch in einfachen Echtzeitanwendungen oder bei Mikrocontrollern üblich
(z.B. Steuerung einfacher Maschinen).

Notizen

Einprogrammbetrieb

Hochschule RheinMain

Nur ein Prozess befindet sich zu einem Zeitpunkt im Speicher.
Zuweisung des gesamten Speichers an diesen Prozess.

Einfache Mikrocomputer: Speicheraufteilung zwischen einem Prozess und dem Betriebssystem.

Typische Alternativen[3]:

[3] ROM = Read Only Memory, RAM = Random Access Memory (read/write)

Notizen

Beispiel: IBM PC (i8088)

Notizen

Mehrprogrammbetrieb

Ziel: Bessere Ausnutzung der CPU (vgl. Kap. 1).

Aufteilung des Speichers in feste Partitionen bei Systemstart. In jede Partition kann ein Programm geladen werden.

Beispiel: IBM Mainframe /360 Betriebssystem OS/MFT (Multiprogramming with a Fixed number of Tasks).

Programme können für eine Partition gebunden sein und sind dann nur in dieser lauffähig.

Notizen

„Krudes“ Beispiel: MS-DOS

Hochschule RheinMain

„Mehrprogrammbetrieb“ mit TSR[4]-Programm

[4] terminate, stay resident

Notizen

Mehrprogrammbetrieb (2)

Hochschule RheinMain

Freie Auswahl einer Partition verlangt Lösung des Relokationsproblems (Verschiebbarkeit, s.o.), d.h. Anpassung aller (absoluten) Adressen des Programms in Abhängigkeit von der Ladeadresse.

Notizen

Mehrprogrammbetrieb (3)

Hochschule RheinMain

Mehrprogrammbetrieb führt auch zum **Schutzproblem**:
Programme können aufgrund der absoluten Adressierung Speicherbereiche anderer Benutzer lesen und schreiben (unerwünscht !).

Lösung des Schutzproblems z.B. in IBM /360:

- Jedem 2kB-Block des Arbeitsspeichers wird ein 4-Bit Schutzcode (*protection key*) als Schloss zugeordnet.
- Jeder Prozess besitzt einen ihm im Programmstatuswort (PSW) zugeordneten Schlüssel, der beim Zugriff auf einen Block passen muss, ansonsten: → Abbruch (SIGSEGV).
- Nur das Betriebssystem kann Schutzcodes von Speicherblöcken und Schlüssel von Prozessen ändern.

⇒ Fazit: Ein Benutzerprozess kann weder andere Benutzerprozesse noch das Betriebssystem stören.

Notizen

Mehrprogrammbetrieb (4)

Hochschule RheinMain

Alternative Vorgehensweise zur Lösung des **Schutzproblems und des Relokationsproblems**:
Ausstattung der CPU mit zwei zusätzlichen Registern, die **Basisregister** und **Grenzregister** genannt werden (*base and bound register*).

Weiterer Vorteil eines Basisregisters zur Relokation: Verschieblichkeit des Programms **nach Programmstart** wird möglich.

Notizen

Swapping

Hochschule RheinMain

Einführung

Bei klassischen Timesharing-Systemen gibt es normalerweise nicht genügend Hauptspeicher, um Prozesse aller Benutzer aufzunehmen.

Swapping beinhaltet das Verschieben von Prozessen vom Hauptspeicher auf Sekundärspeicher (Auslagern) und umgekehrt (Einlagern).

Im Hauptspeicher zu jedem Zeitpunkt nur eine Teilmenge der Prozesse, diese sind aber **vollständig repräsentiert** (im Gegensatz zu virtuellem Speicher, vgl. 9.5).

Die restlichen (möglicherweise auch rechenwilligen) Prozesse werden in einem Bereich im Hintergrundspeicher abgelegt. Dieser Bereich heißt **Swap-Bereich** (*swap area*).

Beispiel: Im Unix vor Einführung der virtuellen Speicherverwaltung genutzt.

Notizen

Variable Partitionen

Hochschule RheinMain

Definition: variable Partitionen

Mehrprogrammbetrieb kann in **variablen Partitionen** erfolgen, d.h.: Anzahl, Anfangsadresse und Länge der Partitionen und damit der eingelagerten Prozesse ändern sich dynamisch.

Definition: Segment

Ein zusammenhängender Speicherbereich variabler Länge heißt auch **Speichersegment** oder einfach **Segment**.

Ziele variabler Partitionen:

- Anpassung an die tatsächlichen Speicheranforderungen
- Vermeidung von Verschwendung.

Notizen

Beispiel

Notizen

Speicherverdichtung

Definition: Externe Fragmentierung

Zerstückelung von Freispeicher zwischen den belegten Segmenten wird **externe Fragmentierung** genannt.

Abhilfe: Alle unbelegten Speicherbereiche können durch Verschiebung der belegten Segmente zu einem einzigen Bereich zusammengefasst werden. (⇒ **Speicherverdichtung** oder **Kompaktifizierung**).

Beispiel:

Notizen

Dynamische Speicheranforderungen (1)

Hochschule RheinMain

Speicherbedarf eines Prozesses kann sich im Verlaufe ändern (i.d.R. wachsen)

Prozess stellt **dynamische Speicheranforderungen**, z.B. zur Aufnahme von neu erzeugten Objekten auf der Halde (dem Heap).

Behandlung dynamischer Speicheranforderungen:

- Belegen angrenzender „Löcher“, falls vorhanden.
- Verschiebung von anderen Prozessen, falls möglich.
- Auslagern von anderen Prozessen.
- „Raum zum Wachsen“: Bei der ersten Speicherbelegung vorab zusätzlichen Speicherplatz allokieren.

Notizen

Dynamische Speicheranforderungen (2)

Hochschule RheinMain

Beispiele für „Raum zum Wachsen“:

(a) Wachsender Datenbereich eines Prozesses

(b) Wachsender Datenbereich und wachsender Stack

Notizen

Zusammenfassung: Swapping

Hochschule RheinMain

Im Hauptspeicher wird zu jedem Zeitpunkt nur eine Teilmenge der Prozesse gehalten. Die restlichen Prozesse sind in einen Swap-Bereich auf Platte ausgelagert.

Alle Prozesse im Hauptspeicher sind vollständig repräsentiert.

Notizen

Algorithmen der Speicherverwaltung

Hochschule RheinMain

Verfahren zur Freispeicherverwaltung

- Speicherverwaltung mit Bitmaps (9.4.1)
- Speicherverwaltung mit verketteten Listen (9.4.2)
- Speicherverwaltung mit dem Buddy-System (9.4.3)

Notizen

Speicherverwaltung mit Bitmaps

Arbeitsweise

Der zur Verfügung stehende Speicher wird in Abschnitte fester Länge unterteilt (Granularität einige Worte bis einige KB).
Jedem Abschnitt wird ein Bit in einer Bitmap zugeordnet:
0 → Einheit ist frei, 1 → Einheit ist belegt.
Je kleiner die Einheit desto größer die Bitmap.
Je größer die Einheit desto mehr Speicher wird verschwendet, da die letzte „angebrochene“ Einheit voll allokiert werden muss.
Hauptproblem: Belegen eines Speicherbereichs von k Einheiten erfordert Durchsuchen der Bitmap nach einer Folge von k Null-Bits → Aufwändig!
Für Hauptspeicherverwaltung daher selten eingesetzt.

Notizen

Speicherverwaltung mit verketten Listen (1)

Arbeitsweise

Jedem belegten und jedem freien Speicherbereich wird ein Listenelement zugeordnet.
Segmente dürfen variabel lang sein.
Jedes Listenelement enthält Startadresse und Länge des Segments, sowie den Status (B=belegt, F=frei).
Gefundenes freies Segment wird (falls zu groß) aufgespalten.
Freigegebenes Segment wird ggf. mit ebenfalls freien Nachbarsegmenten „verschmolzen“.

Notizen

Speicherverwaltung mit verketten Listen (2)

Die freien Segmente können auch in separater Liste geführt werden („Freiliste“). Die Freiliste kann „in sich selbst“ gehalten werden, d.h. in den verwalteten freien Bereichen. (→ kein zusätzlicher Speicher notwendig).
Die Segmentliste kann nach Anfangsadressen geordnet sein.
Vorteil: freiwerdendes Segment kann mit benachbartem freien Bereich zu *einem* freien Segment „verschmolzen“ werden.
Freiliste kann alternativ nach der Größe des freien Bereichs geordnet sein.
Vorteil: Vereinfachung beim Suchen nach einem freien Bereich bestimmter Länge.

Notizen

Strategien zur Speicherallokation

Gegeben: Speicheranforderung gegebener Größe (z.B. `malloc(groesse)`)
Aufgabe: Finde ein passendes Segment in der Freiliste.
Voraussetzung: Liste der Segmente nach Anfangsadressen geordnet.

Definitionen

First Fit: Durchsuche Freiliste, bis ein freies Segment hinreichender Größe gefunden ist. Zerlege den freien Bereich in ein Segment der geforderten Größe und ein neues freies Segment für den übrig bleibenden Rest. Kurze Suchzeit.
Rotating First Fit oder **Next Fit**: Variante von First Fit. Anstelle von vorn zu suchen, beginne Suche die an der nachfolgenden Stelle, an der beim letzten Durchlauf ein Segment belegt wurde.
Best Fit: Durchsuche die gesamte Liste, wähle das kleinste für die Anforderung gerade ausreichende freie Segment und spalte dieses. Langsam. Neigt zur ungewollten Erzeugung vieler kleiner Freisegmente (*Fragmentierung*).
Worst Fit: Wähle das größte freie Segment und spalte dieses. Vermeidet Entstehung vieler kleiner Freisegmente, aber keine gute Idee.

Notizen

Analyse (1)

Hochschule RheinMain

Speicherverwaltung mit Bitmaps oder verketteten Listen führt zu **externer Fragmentierung**.

50%-Regel

50%-Regel (Knuth): Sind im Mittel n Prozesse im Speicher, so gibt es im Mittel $\frac{n}{2}$ freie Bereiche.

Ursache ist, dass zwei benachbarte Freibereiche zu einem zusammengefasst werden, belegte Bereiche hingegen nicht.

→ Im Mittel gibt es doppelt so viele belegte Bereiche, wie Freibereiche.

Notizen

Analyse (2)

Hochschule RheinMain

Damit...

Ungenutzte-Speicher-Regel

Ungenutzte-Speicher-Regel: Sei s die mittlere Größe eines Prozesses, und $k \cdot s$ sei die mittlere Größe eines Freibereichs für einen Faktor k, der abhängig vom Allokationsalgorithmus ist.
Dann gilt für den ungenutzten Anteil f des Speichers:

$$\begin{aligned} f &= \frac{ungenutzt}{gesamt\ belegt} \\ f &= \frac{\frac{n}{2} \cdot k \cdot s}{\frac{n}{2} \cdot k \cdot s + n \cdot s} \\ f &= \frac{k}{k+2} \end{aligned}$$

Beispiel: Sei $k = 0,5$. Dann werden 20% des Speicherplatzes für Freibereiche verbraucht.

Notizen

Speicherverwaltung mit dem Buddy-System

Hochschule RheinMain

Arbeitsweise:

Freie und belegte Speicherbereiche haben ausschließlich Längen, die 2er Potenzen sind.

Da nur bestimmte Längen auftreten, wird von **Speicherblöcken** (anstelle von Segmenten) gesprochen.

Die maximale Blocklänge L_{max} entspricht der größten 2er Potenz kleiner oder gleich dem verfügbaren Speicher, z.B. $L_{max} = 2^{22}$ bei 4 MB oder 6 MB[5]

Es kann eine minimale Blocklänge größer $2^0 = 1$ Byte geben, z.B. $L_{min} = 2^6 = 64$ Bytes.

Eine Speicheranforderung wird auf die nächst mögliche 2er Potenz aufgerundet, z.B.: angefordert 70 KB, belegt 128 KB

Für jede der verwalteten Blocklängen zwischen L_{min} und L_{max} wird eine separate Freiliste gehalten.

[5] 4 MB= $4096 \cdot 1024 = 2^{12} \cdot 2^{10} = 2^{22}$.

Notizen

Buddy-System Arbeitsweise (2)

Hochschule RheinMain

Kein freier Block der gesuchten Länge: Halbiere einen Block der nächsthöheren 2er Potenz in 2 freie Blöcke der gesuchten Länge. Diese werden ***Buddies*** genannt (deutsch: „Kumpel“).
Freigabe eines Blocks der Länge 2^k: wenn sein Buddy bereits ein freier Block ist, mit diesem zu einem neuen freien Block der Länge 2^{k+1} verschmelzen.
Es lassen sich **nicht** zwei beliebige benachbarte Blöcke gleicher Länge 2^k zusammenfassen!
Die Entscheidung ist einfach aufgrund der Adresse der freien Blöcke möglich:

Notizen

Buddy-System Arbeitsweise: Beispiel

Notizen

Fazit Buddy-System

Vorteile:

Bei Speicheranforderung insgesamt schnelles Auffinden oder Erzeugen eines freien Blocks.

Bei Freigabe muss nur *eine* Freiliste durchsucht werden. Falls Buddy bereits frei ist, einfaches Verschmelzen zu einem größeren Block.

Nachteile

Ineffizient in der Speicherausnutzung, da immer auf die nächste 2er Potenz aufgerundet werden muss.

Diese Verschwendung wird als **interne Fragmentierung** bezeichnet, da sie innerhalb des allokierten Speicherbereichs auftritt (im Gegensatz zur **externen Fragmentierung**).

Notizen

Zusammenfassung: Speicherverwaltungsalg.

Hochschule RheinMain

Es wurden Verfahren zur Freispeicherverwaltung besprochen:

- Verwaltung mittels Bitmaps
- Verwaltung mittels verketteter Listen
- Buddy-System

Es wurden Strategien zur Speicherallokation besprochen:

- First Fit
- Rotating First Fit / Next Fit
- Best Fit

Wichtige Begriffe und Regeln:

- Externe und interne Fragmentierung
- 50% Regel, ungenutzte Speicher Regel

Notizen

Virtueller Speicher

Hochschule RheinMain

Problem:

Die Größe eines einzelnen Prozesses kann die Größe des insgesamt zur Verfügung stehenden Speichers übersteigen.

→ Swapping (d.h. Auslagern ganzer Prozesse) ist keine Lösung mehr.

Historische Lösung: *Overlays* (Überlagerungen)

Funktionsteile werden bei Bedarf (D.h. auf explizite Anfrage des Prozesses) dynamisch vom Hintergrundspeicher nachgeladen.
Overlay-Struktur des Programms wird vom Programmierer vorgeplant und vom Binder erzeugt.

heute z.T. wieder aktuell bei Echtzeitsystemen (wegen planbarer Nachlade-Zeitpunkte)

Notizen

Modernerer Ansatz: Virtueller Speicher

Hochschule RheinMain

Den Prozessen wird die Existenz eines großen Speicherraums vorgespiegelt. Dieser **virtuelle Speicher** wird vom Betriebssystem auf Basis von realem Hauptspeicher und Hintergrundspeicher realisiert.
Das Betriebssystem hält dazu die „gerade in Benutzung“ befindlichen Teile eines Programms im Hauptspeicher, Rest auf dem Hintergrundspeicher.
Ausgelagerte Teile werden bei Bedarf (*on Demand*) eingelagert.
Für das Programm ist der Unterschied nicht erkennbar, daher sind keine besonderen Vorkehrungen bei der Programmierung nötig.

Definition: virtueller Adressraum

Die von den Instruktionen eines Programms generierten Adressen werden als **virtuelle Adressen** bezeichnet. Sie bilden zusammen den **virtuellen Adressraum** des Prozesses, der das Programm ausführt.

Zwei Ansätze:

- **Paging**: eindimensionaler virtueller Speicher. Heute am stärksten verbreitet.
- **Segmentierung**: zweidimensionaler virtueller Speicher.

Notizen

Speicherverwaltungseinheit (MMU)

Hochschule RheinMain

Defintion MMU

Die **Speicherverwaltungseinheit** (*Memory Management Unit*, **MMU**) wandelt virtuelle Adressen in reale (physikalische) Adressen um.

Abbildung wird abstrakt auch als **Speicherfunktion** bezeichnet

$$\underset{\text{virtuelle Adresse}}{A_{virt}:} \longrightarrow \boxed{\begin{array}{c}\text{MMU}\\ A_{phys} = S(A_{virt})\end{array}} \longrightarrow \underset{\text{physikalische Adresse}}{A_{phys}:}$$

N.B. $A_{virt} \rightarrow A_{phys}$ ist eindeutig, die **Umkehrung aber nicht**!

Notizen

Technische Umsetzung

Nur die höherwertigen $n - m$ Adressbits[6] werden umgewandelt:

⇒ Alle virtuellen Adressen, die sich nur in ihren m niederwertigen Adressbits unterscheiden, gehören gemeinsam zu einer **Seite** und werden durch die MMU auf denselben **Seitenrahmen** abgebildet.

Die Größe der Seiten sowie der Seitenrahmen ist dabei 2^m.
(im Beispiel: $2^{12} = 4096$)

[6] Beispiel hier: $A_{12} \ldots A_{63} \rightarrow n - m = 52,\ n = 64, m = 12$

Notizen

Grundbegriffe des Paging

Definition: Seiten, Seitenrahmen

Der virtuelle Adressraum ist in Einheiten fester Länge unterteilt, diese heißen **Seiten (*Pages*)**.
Der physikalische Speicher wird ebenfalls in Einheiten dieser Länge unterteilt, diese heißen **Kacheln** oder **Seitenrahmen (*Page Frames*)**.

Der Transfer zwischen Hauptspeicher und Hintergrundspeicher erfolgt grundsätzlich in Einheiten von Seiten.

Heute typische Größen:

Virtueller Adressraum:

- 2^{32} Bytes (4 GB) (32-bit x86)
- 2^{48} Bytes (256 TB) (64-bit AMD64, Intel64)

Physikalischer Speicher:

- i.d.R. kleiner (z.B. 4 - 16 GB für Arbeitsplatzrechner)

Seiten / Seitenrahmen: 4 KB oder 8 KB

Notizen

Grundbegriffe des Paging (2)

Hochschule RheinMain

Definition: Seitentabelle

Die abstrakte Speicherfunktion $S()$ kann bildlich durch eine Tabelle repräsentiert werden, bezeichnet als **Seitentabelle**.

Die Seitentabelle ordnet jeder Seite des virtuellen Adressraums einen Seitenrahmen des Hauptspeichers **oder** ein Kennzeichen zu, dass die Seite sich nicht im Hauptspeicher befindet (nicht eingelagert ist). Dieses Kennzeichen wird **Present/Absent-Bit** genannt.

Dereferenziert ein Programm eine virtuelle Adresse, deren zugehörige Seite nicht eingelagert ist, so erzeugt die MMU eine Unterbrechung der CPU. Diese Situation wird als **Seitenfehler (*Page Fault*)** bezeichnet.

Notizen

MMU-Funktionsprinzip

Hochschule RheinMain

Beispiel: MMU, 16 Seiten zu 4 KB

Notizen

Seitentabelleneintrag

Neben der Seitenrahmennummer und dem Present/Absent-Bit („P“) enthält ein Seitenrahmentabelleneintrag in der Regel weitere Informationen:

- R Referenced-Flag: auf die Seite wurde lesend und / oder schreibend zugegriffen
- M Modified-Flag: Seite wurde verändert; „dirty bit“
- A Access Rights: Zugriffsrechte: Seite ist lesbar / schreibbar / ausführbar
- Weitere (z.B. non-cacheable (C), cache write through (W), ...)

Notizen

Behandlung eines Seitenfehlers (1)

Behandlung eines Seitenfehlers besteht prinzipiell in der **Einlagerung der fehlenden Seite** vom Hintergrundspeicher in einen freien Seitenrahmen des Arbeitsspeichers.
Einlagerung als Reaktion auf einen Seitenfehler wird als ***Demand Paging*** bezeichnet.
Liegt kein freier Seitenrahmen vor, muss ein solcher gewonnen werden.
I.d.R. hält das Betriebssystem einen Vorrat an freien Rahmen vor.
(Algorithmen für die Auswahl zu ersetzender Seiten werden detailliert in 9.6 besprochen).
Falls die zu ersetzende Seite während ihrer Lebenszeit im Speicher verändert wurde (als *Dirty Page* bezeichnet), wird sie auf die Platte zurückgeschrieben (*Update*). Der belegte Seitenrahmen ist danach frei.
Falls die zu ersetzende Seite nicht verändert wurde (enthält z.B. Programmcode), ist kein Zurückschreiben erforderlich.

Notizen

Behandlung eines Seitenfehlers (2)

Notizen

Probleme mit Seitentabellen

Die Seitentabelle kann sehr groß werden, z.B.:

Seitengröße: 4 KB ($= 2^{12}$ Bytes)
32-Bit Architektur:

- D.h. Adressen sind 32 Bit lang.

⇒ Seitentabelle braucht $\frac{2^{32}}{2^{12}} = 2^{20} = 1048576$ Einträge à 4 Byte $\hat{=}$ 4 MB!

- **und**: jeder Prozess hat seinen eigenen Adressraum, damit auch seine eigene Seitentabelle!

64-Bit Architektur:

- D.h. Adressen sind 64 Bit lang.

⇒ Seitentabelle braucht $\frac{2^{64}}{2^{12}} = 2^{52} \approx 4.5 \cdot 10^{15}$ Einträge!

→ **Probleme:**

Hoher Platzbedarf zum Halten der Seitentabelle (s.o.)
Jede Operation des Prozessors (Befehl holen, Daten lesen / schreiben) erfordert eine Referenzierung der Seitentabelle → Langsam!

Lösungen (im Folgenden diskutiert):

Platzproblem: Mehrstufige Seitentabellen
Geschwindigkeitsproblem: Translation Lookaside Buffer (TLB)

Notizen

Mehrstufige Seitentabellen

Prozesse belegen in der Regel nicht den gesamten Adressraum

viele Seitentabelleneinträge bleiben also leer (P = 0).

verschwenden damit selbst Speicherplatz.

Lösung: Mehrstufige Seitentabellen (*wieso* spart das Speicher?)

Quelle: AMD64 Architecture Programmer's Manual, Vol. 2

Seitennummer (hier: 20-Bit) wird aufgeteilt in 2 x 10-Bit:

Page Directory Offset: Index in 1. Tabellenstufe, führt zu 2. Stufe

Page Table Offset: Index in 2. Stufe, führt zu Seitentabelleneintrag

Notizen

Translation Lookaside Buffer (TLB)

Prozesse zeigen i.d.R. Lokalitätsverhalten

→ Einige (wenige) Seitentabelleneinträge werden immer wieder verwendet.

Lösung: TLB: Cache für Seitentabelleneinträge

Schneller Assoziativspeicher, in MMU integriert
Hält kleinere Anzahl der zuletzt genutzten Seitentabelleneinträge vor
Bei TLB Hit: Seitentabelleneintrag wird schnell geliefert
Bei TLB Miss: „Page Table Walk“, gefundener Seitentabelleneintrag wird gespeichert
Dabei Verdrängungsstrategie: z.B. LRU oder Random

→ **Im Regelfall große Beschleunigung, aber zu beachten:**

Bei Prozessumschaltung: TLB muss ggf. vom Betriebssystem gelöscht werden
Nach Prozessumschaltung: TLB nachladen kostet zusätzliche Zeit

→ Zu häufige Prozesswechsel bewirken viel Overhead
(Vgl. 4.3.1, processor sharing für Grenzfall $q \longrightarrow 0$)

Notizen

Seitenersetzungsalgorithmen

Hochschule RheinMain

Problem:

Geschickte Auswahl von zu ersetzenden Seiten des Hauptspeichers:

Wird eine häufig benutzte Seite ausgelagert, ist die Wahrscheinlichkeit groß, dass diese schon bald wieder eingelagert werden muss, was Zusatzaufwand für das Betriebssystem und Blockierungszeiten für den betroffenen Prozess mit sich bringt.

Ziel:

Gute Performance des Systems.

„Günstig“ auszulagernde Seiten *schnell* identifizieren.

Seitenersetzungsalgorithmen (*Page Replacement Algorithms*) sind ein klassisches Problem der Betriebssysteme, vielfach theoretisch und experimentell untersucht.

Notizen

Überblick

Hochschule RheinMain

In der BS-Literatur betrachtete Seitenersetzungsverfahren:

Der optimale Seitenersetzungsalgorithmus

Not-Recently-Used (NRU)

First-In, First-Out (FIFO) und Second-Chance

Clock-Algorithmus

Least-Recently-Used (LRU)

Aging-Algorithmus

Notizen

„Optimales“ Verfahren

Hochschule RheinMain

Algorithmus:

Wird ein Seitenrahmen benötigt, bestimme für alle eingelagerten Seiten, wieviele Instruktionen lang die Seite **zukünftig** durch die noch auszuführenden Instruktionen nicht referenziert werden wird.

Wähle die Seite zur Ersetzung aus, die am längsten nicht referenziert werden wird.

Bemerkungen:

Der Algorithmus schiebt den Seitenfehler, der dazu führt, dass die auszulagernde Seite wieder eingelagert werden muss, am weitesten in die Zukunft.

Die benötigte Information über das **zukünftige Programmverhalten** ist aber i.d.R. nicht vorhanden!

Der Algorithmus ist daher nicht realisierbar, nur für Vergleichszwecke interessant.

Notizen

Not-Recently-Used (NRU)

Hochschule RheinMain

Voraussetzungen:

Verwendung der im Seitentabelleneintrag gespeicherten Status-Bits je Seite (vgl.9.5.2):

- **R-Bit** (*referenced*): wird gesetzt, wenn auf die Seite **lesend oder schreibend** zugegriffen wird.
- **M-Bit** (*modified*): wird gesetzt, wenn auf die Seite **schreibend** zugegriffen wird).

Status-Bits werden durch die Hardware automatisch gesetzt.

Gesetzte Status-Bits können durch das Betriebssystem (softwaremäßig) zurückgesetzt werden.

Notizen

Algorithmus

Bei Prozessstart werden beide Status-Bits für alle Seiten des Prozesses zurückgesetzt.
Periodisch, z.B. bei jeder Uhr-Unterbrechung, wird das Referenced-Bit aller Seiten durch das Betriebssystem zurückgesetzt. Dadurch können im folgenden referenzierte Seiten von nicht referenzierten unterschieden werden.
Uhr-Unterbrechungen löschen das Modified-Bit **nicht**. Es wird weiter für die Entscheidung über das Zurückschreiben benötigt.
Tritt ein Seitenfehler auf, teile alle eingelagerten Seiten in die folgenden Klassen entsprechend der aktuellen Werte des R- und des M-Bits ein:

- Klasse 0: nicht referenziert, nicht modifiziert.
- Klasse 1: nicht referenziert, modifiziert.
- Klasse 2: referenziert, nicht modifiziert.
- Klasse 3: referenziert, modifiziert.

Wähle zufällig aus der kleinst-nummerierten nicht-leeren Klasse eine Seite zur Ersetzung aus.

Notizen

Bemerkungen

Der Not-Recently-Used-Algorithmus ist ...

...Einfach

...effizient implementierbar

...nicht optimal, aber Performance ist akzeptabel

Notizen

First-In, First-Out (FIFO)

Hochschule RheinMain

Idee: Die zuerst eingelagerte Seite wird auch zuerst wieder ausgelagert

Verwaltung über eine Liste:

- Bei Einlagerung wird Eintrag am Listenende angehängt
- Bei Seitenfehler: Auszulagernde (älteste) Seite steht im Listenkopf, Listenkopf wird danach entfernt.

Eher ungeschicktes Verfahren (die älteste Seite kann trotzdem ständig gebraucht werden, es würde dann bald wieder ein Seitenfehler für diese Seite erzeugt).

Notizen

Second Chance

Hochschule RheinMain

Idee: Ähnlich FIFO, aber mit Beachtung des Referenced-Bits

Bei Seitenfehler: Vom Listenkopf ausgehend Seiten-Knoten durchlaufen:

- Wenn R-Bit gesetzt: dieses löschen und Seite hinten anhängen (→ Seite erhält eine „zweite Chance“)
- Wenn R-Bit nicht gesetzt: Seite auslagern, fertig

Sollte überall das R-Bit gesetzt sein, degeneriert das Verfahren zu FIFO (erster Eintrag ist mit gelöschtem R-Bit wieder vorne)

Notizen

Clock-Algorithmus

Hochschule RheinMain

Alle Seiten werden als zyklische Liste (in Form einer Uhr) verwaltet

Der Zeiger der Uhr zeigt auf die älteste Seite.

Bei Seitenfehler: älteste Seite prüfen

Wenn R-Bit gesetzt: dieses löschen, Zeiger weiterrücken, wiederholen

Wenn R-Bit nicht gesetzt:

- Seite auslagern
- einzulagernde Seite einfügen
- Zeiger weiterrücken

Der Clock-Algorithmus ist eine bzgl. der Implementierung verbesserte Form des Second-Chance Algorithmus.

Notizen

Least-Recently-Used (LRU)

Hochschule RheinMain

Voraussetzungen:

Gute Annäherung an den optimalen Algorithmus ist möglich, wenn man aus dem Zugriffsverhalten in der jüngeren Vergangenheit auf das Verhalten in der Zukunft schließen kann.

Beobachtung: Seiten, die schon lange nicht mehr benötigt wurden, werden auch mit hoher Wahrscheinlichkeit für längere Zeit nicht mehr benötigt werden.

Algorithmus:

Wenn ein Seitenfehler auftritt, ersetze die Seite, die am längsten unbenutzt ist (*least-recently-used*, LRU).

Notizen

Least-Recently-Used (LRU) (2)

Hochschule RheinMain

Bemerkungen:

LRU ist realisierbar, Implementierung ist aber aufwändig.

Implementierungsidee:

- Die Menge aller Seiten im Speicher wird mit einer verketteten Liste verwaltet.
- Die gerade benutzte Seite steht am Kopf der Liste, die am längsten nicht benutzte Seite am Ende.
- Bei jedem Speicherzugriff wird die zugegriffene Seite in der Liste gesucht und an den Anfang gestellt (Umordnung!).
- Wähle die Seite am Ende der Liste als zu ersetzende Seite.

Eine effiziente Implementierung ist nur mit spezieller Hardwareunterstützung möglich und nur in Ausnahmefällen sinnvoll (z.B. für die Verwaltung von TLBs mit wenigen Einträgen).

Notizen

Aging-Algorithmus (Alterung)

Hochschule RheinMain

Ziel:

Approximation von LRU in Software.

Algorithmus:

Jeder Seite wird Software-Zähler der Länge L Bit zugeordnet (z.B. 8-Bit lang), initialisiert mit 0.

Bei jeder Uhr-Unterbrechung für jede eingelagerte Seite:

- Den Zähler der Seite wird um 1 Bit nach rechts schieben (Aging = Altern).
- Das R-Bit der Seite ersetzt die führende Stelle des Zählers.
- Das R-Bit wird gelöscht. (Es wird durch die Hardware im Falle eines weiteren Zugriffs automatisch neu gesetzt).

Wenn ein Seitenfehler auftritt, wird die Seite mit dem kleinsten Zählerwert ersetzt.

Notizen

Aging: Beispiel

Hochschule RheinMain

R-Bits und Zähler der Seiten $0 \ldots 5$ **nach** $t = 1 \ldots 6$ **Uhr-Unterbrechungen**

t = 1

Seite	R	Zähler
0	1	10000000
1	0	00000000
2	1	10000000
3	0	00000000
4	1	10000000
5	1	10000000

t = 2

Seite	R	Zähler
0	1	11000000
1	1	10000000
2	0	01000000
3	0	00000000
4	1	11000000
5	0	01000000

t = 3

Seite	R	Zähler
0	1	11100000
1	1	11000000
2	0	00100000
3	1	10000000
4	0	01100000
5	1	10100000

t = 4

Seite	R	Zähler
0	1	11110000
1	0	01100000
2	0	00010000
3	0	01000000
4	1	10110000
5	0	01010000

t = 5

Seite	R	Zähler
0	0	01111000
1	1	10110000
2	1	10001000
3	0	00100000
4	0	01011000
5	0	00101000

t = 6

Seite	R	Zähler
0	1	10111100
1	0	01011000
2	1	11000100
3	0	00010000
4	0	00101100
5	0	00010100

Kleinster Zähler: 0

→ Seite 3 auslagern!

Notizen

Aging: Bemerkungen

Hochschule RheinMain

Bemerkungen zum Verhältnis zu LRU:

Information über das Zugriffsverhalten auf eine Seite während eines Uhr-Intervalls wird auf ein einziges Bit vergröbert.

Information, die älter als L Zeitintervalle ist, wird als wertlos eingestuft (geht durch Rechts-Shifts verloren).

Eine Seite, die während n Uhr-Unterbrechungen nicht referenziert wurde, besitzt n führende Nullen im Zähler.

Die Auswertung des Zählers muss nur nach Ablauf dieses Zeitintervalls erfolgen, nicht bei jeder Speicherreferenz.

Die Aufzeichnung von nur einem Bit je Zeitintervall erlaubt damit nicht die Unterscheidung einer früheren oder einer späteren Referenz innerhalb des Intervalls.

Insgesamt ist die Approximation von LRU hinreichend gut.

Notizen

Entwurfsprobleme bei Paging-Systemen

Notizen

Ziel:

Kennenlernen von Aspekten, die beim Entwurf von Paging-Systemen besondere Beachtung finden, da sie starken Einfluss auf die Performance haben.

Überblick:

Working Set Modell

Lokale oder globale Seitenersetzung

Seitengröße

Implementierungsprobleme

Das Working Set Modell (1)

Notizen

Prozesse zeigen i.d.R. ein **Lokalitätsverhalten**.

Definition: Working Set

Die Menge der Seiten, die ein Prozess augenblicklich nutzt, wird sein **Working Set** (**Arbeitsbereich**) genannt.
(P. Denning, 1968).

Ist der zugeordnete Hauptspeicher zu klein, um das Working Set aufzunehmen, so entstehen sehr viele Page Faults.

Wenn nach wenigen Zugriffen die Seite wieder benötigt wird, die gerade verdrängt wurde, so sinkt die Ausführungsgeschwindigkeit auf einige Instruktionen je Seiteneinlagerung (typisch mehrere Millisekunden). Man bezeichnet eine solche Situation als **Thrashing**.

Das Working Set Modell (2)

Hochschule RheinMain

Einlagern von Seiten, obwohl noch keine entsprechenden Seitenfehler vorliegen, wird als **Prepaging** bezeichnet (im Gegensatz zum üblichen **Demand Paging**).

Ist das Working Set eines Prozesses bekannt, kann mittels Prepaging nach einem Prozesswechsel verhindert werden, dass der Prozess durch eine große Anzahl von Seitenfehlern seine Umgebung zunächst wieder aufbauen muss, in dem das Working Set vollständig eingelagert wird. (Working Set Model, Denning, 1970)

Durch das Working Set Modell kann die Seitenfehlerrate drastisch gesenkt werden.

Notizen

Das Working Set Modell (3)

Hochschule RheinMain

Ansatz zur Bestimmung des Working Sets:

Verwendung des Aging-Algorithmus (siehe 9.6.6).

Eine 1 in den höherwertigen n Bits des Alterungszählers kennzeichnet die zugehörige Seite als zum Working Set gehörig.

Wird eine Seite für n Uhrticks nicht referenziert, so scheidet sie aus dem Working Set aus.

Verbesserung des Clock-Algorithmus (siehe 9.6.4) ist möglich:

Seiten, die zum Working Set gehören, werden bei gelöschtem R-Bit übersprungen.

Dieser Algorithmus wird **wsclock** (*Working Set Clock*) genannt.

Notizen

Mindestgröße

Hochschule RheinMain

Wie viele Seiten kann eine einzige Maschineninstruktion referenzieren?

Befehlscode lesen

Quelloperand holen

Zieloperand speichern

Operand/Maschineninstruktion i.d.R. $>$ 1 Byte

→ Jeder kann (worst-case) über max. 2 Seiten verteilt sein.

⇒**Mindestens 6 Seitenrahmen[7] für einen Prozess, sonst u.U. „Thrashing innerhalb einer Instruktion“!**

[7] Architekturabhängig: z.B. MIPS: nur 2 Seitenrahmen

Notizen

Lokale oder globale Seitenersetzung

Hochschule RheinMain

Im Mehrprogrammbetrieb haben mehrere Prozesse Seiten im Speicher.

Lokale Seitenersetzung: bei der Auswahl einer zu ersetzenden Seite werden nur die Seiten des Prozesses selbst betrachtet.

Globale Seitenersetzung: alle im Speicher befindlichen Seiten, also auch die anderer Prozesse, werden in Betracht gezogen.

Lokale Seitenersetzungs-Algorithmen ordnen jedem Prozess eine feste Anzahl von Seitenrahmen zu, globale Algorithmen alloziieren Seitenrahmen dynamisch.

Globale Algorithmen arbeiten i.A. mit **besserer Performance**.

Im Falle eines globalen Algorithmus muss das Betriebssystem dynamisch entscheiden, wie viele Seitenrahmen jeder Prozess zugewiesen bekommt. Einfacher Ansatz: Betrachtung des Working Set, verhindert aber kein Thrashing.

Notizen

Lokale oder globale Seitenersetzung (2)

Bester Ansatz: Kontrolle der Seitenfehlerfrequenz (Page Fault Frequency PFF) über die Anzahl der zugeordneten Seitenrahmen.
Typischer Verlauf:

Seitenfehlerrate zwischen A und B wird als akzeptabel angesehen. Steigt die Seitenfehlerrate über A, werden zusätzliche Seitenrahmen zugeordnet, sinkt sie unter B, werden Seitenrahmen entzogen.

Notizen

Seitengröße

Bestimmung der optimalen Seitengröße erfordert ein Ausbalancieren von verschiedenen widersprechenden Faktoren:

- Geringe interne Fragmentierung spricht für eine kleine Seitengröße.
- Kleine Seiten erfordern eine große Seitentabelle.
- Kosten für die Ein-/Auslagerung sind stark von Positionierzeiten und Rotationsverzögerung der Platte beeinflusst. Spricht für große Seitengröße.
- Hardware-Aufwand für Memory Management Unit.

Heutige Seitengrößen: 512 B - 8 KB, typisch 4 KB

Notizen

Implementierungsprobleme (1)

Hochschule RheinMain

Instruction Retry nach Seitenfehler:

Eine Instruktion besteht aus mehreren Worten.

Beispiel (MC68000): `MOVE 6(A1), 2(A2)`

Adresse	Speicherinhalt	
0x1000	MOVE x(A1),y(A2)	Opcode
0x1002	6	1. Operand
0x1004	2	2. Operand

Bei Seitenfehler kann PC=0x1000, 0x1002 oder 0x1004 sein

„Wieviel von der Instruktion ist schon abgearbeitet?“

„Wo/wie muss nach Seitenfehler fortgefahren werden?“

Kann kompliziert werden ...

Notizen

Implementierungsprobleme (2)

Hochschule RheinMain

Sperren von Seiten im Speicher

z.B. für Pufferbereiche während gestarteter I/O-Aufträge.

Entsprechende Seiten von blockierten Prozessen dürfen dann nicht ausgelagert werden (reserviert für DMA-Transfer).

Typische Operationen: pin / unpin auf Seiten.

Memory Sharing

Gemeinsam benutzte Seiten verschiedener Prozesse (insbesondere Code-Seiten) müssen berücksichtigt werden.

Auslagern würde Seitenfehler für Partner verursachen.

I.d.R. werden spezielle Datenstrukturen mit Referenzzählern verwendet.

Notizen

Implementierungsprobleme (3)

Hochschule RheinMain

Paging-Dämonen

Definition: Deamon

Ein **Dämon** (*Daemon*) ist ein Hintergrundprozess, der nicht mit einem Benutzer in Interaktion tritt (hat kein Terminal).

Ziel: Vorhalten einer Anzahl freier Seitenrahmen, um bei Auftreten eines Seitenfehlers keine Verzögerung für das Auslagern einer Seite in Kauf nehmen zu müssen (Performance-Aspekt).
Der Page Daemon wird z.B. periodisch aktiviert, um die Speichersituation zu untersuchen:

- Liegen zu wenige freie Rahmen vor, beginnt er, Seiten mithilfe des Seitenersetzungsalgorithmus zur Auslagerung auszuwählen.
- Wurden sie modifiziert, veranlasst er das Zurückschreiben auf den Hintergrundspeicher.

Der ursprüngliche Inhalt als „frei“ geführter Seitenrahmen bleibt bekannt.

- Erfolgt ein erneuter Zugriff auf die ursprüngliche Seite, bevor sie überschrieben wird, wird sie aus dem „Frei“-Pool zurückgewonnen.
- Dieser Vorgang wird **Page Reclaiming** genannt.

Notizen

Segmentierung

Hochschule RheinMain

Ziel

Bisher wurden Paging-Systeme betrachtet, die einen sogenannten eindimensionalen virtuellen Speicher offerieren.

Im folgenden wird sogenannter segmentierter virtueller Speicher besprochen.

Notizen

Begriffsbildung

Hochschule RheinMain

Eindimensionale / Zweidimensionale virtuelle Adressräume:

eindimensional: **zwei**dimensional:

Notizen

Begriffsbildung (2)

Hochschule RheinMain

Definition: zweidimensionale Adresse

Eine zweidimensionale (zweiteilige) virtuelle Adresse besteht aus:

Segmentnummer, die ein Segment selektiert,

Adresse im selektierten Segment (Offset).

Die Menge aller Segmente definiert den sog. Segmentraum.
Ein Segment besitzt einen maximalen linearen Adressbereich $\{0, \ldots, L_{max}\}$. Jedes Segment hat damit die Eigenschaften eines (linearen) Adressraums.
Verschiedene Segmente können unterschiedlich lang sein.
Die Länge jedes Segments liegt zwischen 0 und dem erlaubten Maximum L_{max}.
Die Segmentlänge darf sich i.d.R. während der Lebenszeit des Segments ändern.
Ein Segment ist eine logische Speichereinheit und als solche dem Programmierer bewusst (sollte es zumindest sein).

Notizen

Beispiel zur Nutzung

Hochschule RheinMain

Mögliche Segmente für einen Compiler-Lauf:

Notizen

Segmentierung mit Paging

Hochschule RheinMain

Ein eingelagertes Segment *muss nicht mehr* vollständig im Hauptspeicher sein.

Jedes Segment besteht aus einer Folge von Seiten.

Prinzipielle Adressierung:

Notizen

Beispiel: Speicherverwaltung in UNIX

Hochschule RheinMain

Ziel

Kennenlernen der UNIX-Speicherverwaltung.

Überblick:

Speichermodell.

Systemaufrufe zur Speicherverwaltung.

Notizen

Speichermodell

Hochschule RheinMain

(vgl. 9.1)

Jeder Prozess besitzt einen eigenen virtuellen Adressraum. Der Adressraum enthält aus Benutzersicht (im Anwender-adressbereich) 3 Segmente

- Text-Segment
  - Programmcode (.text)
  - Nur-Lese Daten (.rodata)
- Daten-Segment
  - Initialisierte Daten (.data)
  - Nicht-initialisierte Daten (.bss)
  - Heap (für malloc() & Co.)
- Stack-Segment
  - für lokale Variablen etc.

Notizen

Speichermodell (2)

Hochschule RheinMain

Zusätzlich kann ein Adressraum in neueren UNIX-Systemen ein oder mehrere Segmente der folgenden beiden Typen enthalten:

Gemeinsam benutztes Speichersegment, das gleichzeitig in mehrere Adressräume eingeblendet ist (z.B. als Shared Memory Data Segment oder Shared Library),

Mapped File-Segment zur Aufnahme eines Fensters eines Memory Mapped Files (abgebildete Datei, siehe Kapitel Dateisysteme). Auf ein Mapped File kann ohne explizite E/A-Operationen zugegriffen werden.

Notizen

Adressraumregionen

Hochschule RheinMain

Virtueller Adressraum wird in **Regionen** unterteilt.

Eine Region ist ein zusammenhängender Bereich virtueller Adressen und kann ein identifizierbares Segment (Objekt) aufnehmen (Text-, Daten-, Stack-Segment und andere, s.o.).

Einem zugeordneten Segment entspricht eine Folge von Seiten des Adressraums.

Für jede Region können individuelle Schutzattribute festgelegt werden (read-only, read-write, read-execute). Diese übertragen sich auf die repräsentierenden Seiten.

Gemeinsam genutztes Segment:

- entspricht Regionen in verschiedenen Adressräumen, denen das Segment zugeordnet ist.
- Anfangsadresse der Regionen kann in Adressräumen unterschiedlich sein.
  (**Achtung:** Adressen im gemeinsam benutzten Segment sind dann i.d.R. bedeutungslos).

Notizen

Systemdienste zur Speicherverwaltung

Hochschule RheinMain

Überblick:

Im POSIX-Standard existieren keine expliziten Festlegungen.

Für die Anwendungsprogrammierung wird auf ANSI C-Standardbibliotheksfunktionen zur dyn. Speicherverwaltung zurückgegriffen.

Im folgenden werden betrachtet:

- klassische Dienste zur Behandlung der Standard-Segmente.
- Dienste für Shared Memory Segmente.
- Dienste für Memory-Mapped Files.
- Dienste zur Adressraum-Manipulation.

Dienste für Memory-Mapped Files sowie zur Adressraum-Manipulation sind noch nicht auf allen heutigen UNIX-Varianten verfügbar.

Notizen

Dienste für Standard-Segmente

Hochschule RheinMain

Zur absoluten bzw. relativen Veränderung der Größe des Datensegments (Wachsen oder Schrumpfen) existieren die Systemaufrufe `brk()` bzw. `sbrk()` (letzterer ist häufig eine Bibliotheksfunktion):

```
caddr_t brk(caddr_t addr); caddr_t sbrk(int incr);
```

Auf Programmebene werden i.d.R. die ANSI C-Bibliotheksfunktionen `malloc()` und `free()` benutzt. Implementierung von `malloc()` greift z.B. auf `brk()` zurück, wenn die Heap-Größe erweitert werden muss.

Die Größe des **Stack-Segments** wird nicht auf Programmebene verändert, sondern vom Betriebssystem implizit vergrößert, wenn sich der Stack zur Ausführungszeit als zu klein erweist.

Die Größe des **Text-Segments** ist statisch und damit nicht veränderbar.

Notizen

Dienste für Shared Memory Segmente (1)

Hochschule RheinMain

Shared Memory Segmente sind Bestandteil der System V IPC-Mechanismen und stellen gemeinsam von mehreren Prozessen benutzbare Segmente mit festlegbaren Zugriffsattributen dar.
Die Benutzung von Shared Memory Segmenten setzt i.d.R. die korrekte Synchronisation der Prozesse voraus (z.B. mittels Semaphoren).
Shared Memory Segmente sind i.d.R. die effizienteste Form der Interprozesskommunikation.
Maximale Anzahl von Shared Memory Segmenten im System (`SHMMNI`) und je Prozess (`SHMSEG`) und deren maximale und minimale Größe (`SHMMAX`, `SHMMIN`) werden bei der Konfiguration des BS-Kerns festgelegt.
Shared Memory Segmente überleben, wie die anderen System V IPC-Objekte die sie erzeugenden Prozesse und sind ohne besondere Maßnahmen erst mit einem Neustart des Systems zerstört. Zur Information über die existierenden IPC-Objekte bzw. zu ihrer Zerstörung existieren die Kommandos `ipcs` bzw. `ipcrm`.
(Bitte am Ende jeder Praktikumssitzung zum Aufräumen nutzen !)

Notizen

Dienste für Shared Memory Segmente (2)

Hochschule RheinMain

Operationen:

Header-Dateien: `<sys/types.h>`, `<sys/ipc.h>`, `<sys/shm.h>`

```
int shmget(key_t key, int size, int shmflag)
```

erzeugt oder öffnet ein Shared Memory Segment mit dem externen Schlüssel `key` und liefert den zugehörenden Identifier `shmid` des Segments zurück (oder `-1` bei Fehler). Die Behandlung von `key` und `shmflag` ist insgesamt analog wie bei Semaphoren.
Ist insbesondere `key` auf `IPC_PRIVATE` gesetzt, so wird ein privates Segment (ohne öffentlichen key) erzeugt, das für verwandte Prozesse genutzt werden kann.
Wird ein Schlüssel verwendet und in `shmflag` das `IPC_CREAT`-Bit gesetzt und existiert noch kein IPC-Objekt zu `key`, so wird ein neues Segment der Mindestgröße `size` erzeugt.
Für das Öffnen eines existierenden Objekts darf das `IPC_CREAT`-Flag nicht gesetzt sein. Hat `size` einen Wert größer als die aktuelle Größe des Segments, so schlägt der Aufruf fehl, der Wert `0` akzeptiert das Segment in der aktuellen Größe.
Die Zugriffsrechte für das Segment werden durch einen entsprechenden `mode` in `shmflag` bei der Erzeugung mit festgelegt, typisch `0600` für Lesen und Schreiben durch alle Prozesse des Eigentümers des Segments oder `0666` für beliebige Prozesse. Beim Öffnen eines Segments gibt der Zugriffsmodus die geforderten Rechte an.

Notizen

Dienste für Shared Memory Segmente (3)

```
void * shmat(int shmid, void * addr, int flag)
```

blendet das Segment `shmid` in den Adressraum des Aufrufers an der virtuellen Adresse `addr` ein (attach). Wird für `addr` der Wert `NULL` gewählt, so wird die Startadresse des Segments im Adressraum vom Betriebssystem festgelegt (Konflikt ausgeschlossen), ansonsten an der spez. Adresse, die i.d.R. innerhalb des Anwenderadressbereichs auf Seitengrenze ausgerichtet sein muss. Die getroffene Startadresse wird zurückgegeben (oder `-1` bei Fehler). Ist in `flag` `SHM_RDONLY` gesetzt, so wird das Segment nur mit Leserecht in den Adressraum eingeblendet. `shmat` muss nach Ausführung von `shmget` und vor der eigentlichen Nutzung des Segments ausgeführt werden.

```
int shmdt(void * addr)
```

blendet ein zuvor mittels `shmat` an die Adresse `addr` eingeblendetes Segment wieder aus dem Adressraum des Aufrufers aus (detach). Die Existenz des Shared Memory Segments bleibt hiervon unberührt.

Notizen

Dienste für Shared Memory Segmente (4)

```
int shmctl(int shmid, int cmd, struct shmid_ds * arg)
```

ermittelt / ändert den Status eines Shared Memory Segments oder löscht es. Kommandos (Werte für `cmd`) sind:

- `IPC_STAT` (Abfragen des Status mit Ergebnis in einer Stuktur `shmid_ds`, auf die `arg` zeigt).
- `IPC_SET` (Setzen der Eigentümer-`UID`/`GID` und der Zugriffsrechte bei passender effektiver `UID`).
- `IPC_RMID` (Löschen des Segments bei passender effektiver `UID`; wird effektiv, wenn der letzte Prozess `shmdt` ausführt, zwischenzeitlich ist kein `shmat` mehr möglich, Referenzzähler).
- `SHM_LOCK` (Sperren des Segments, nur durch Prozess mit root-Recht).
- `SHM_UNLOCK` (Entsperren des Segments, nur durch Prozess mit root-Recht).

Notizen

Dienste für Memory Mapped Files (1)

Hochschule RheinMain

Einblenden einer Datei oder eines Gerätes in eine Region des Adressraums
(Kombination von Systemfunktionen der Speicherverwaltung, des I/O-Systems und des Dateisystems)
Eingeblendete, zuvor geöffnete Datei wird wie normales Shared Memory Datensegment zugegriffen, d.h. ohne Nutzung der Dienste `read`, `write`, `lseek`.
Daten werden mithilfe des normalen Paging-Mechanismus aus der hinterlagerten Datei herangeschafft.
Veränderte Seiten der Datei werden irgendwann, bedingt durch Seitenverdrängung, dorthin zurückgeschrieben, wenn der letzte Prozess die zugrundeliegende Datei schließt oder durch `sync`-Vorgänge.

Notizen

Dienste für Memory Mapped Files (2)

Hochschule RheinMain

Vorteile:

- Vermeiden von Kopiervorgängen.
- Weniger Kontext-Umschaltungen, da I/O-Systemaufrufe wegfallen.
- Bequemer Zugriff über Zeiger.
- Veränderungen in der Datei sind für andere Prozesse, die dieselbe Datei eingeblendet haben, *unmittelbar* sichtbar.

Die Anwendbarkeit des Einblendvorgangs für zahlreiche Geräteklassen erlaubt den Zugriff auf Hardware-Komponenten
(z.B. Geräteregister, Graphik-Bildspeicher, usw.), auch aus einem Anwendungsprogramm heraus (hohe Flexibilität, hohe Performance).

Notizen

Dienste für Memory-Mapped Files (3)

Operationen (Überblick):

Header-Dateien: `<sys/types.h>`, `<sys/mman.h>`

```
caddr_t mmap(caddr_t addr, size_t len, int prot,
int flags, int fd, off_t offset)
```

blendet ein Fenster der durch `fd` angegebenen, geöffneten Datei (oder Gerät), beginnend in der Datei an der Stelle `offset`, in der Länge `len` in den Adressraum des Aufrufers an der virt. Adresse `addr` mit der Rechtefestlegung `prot` ein.

```
int munmap(caddr_t addr, size_t len):
```

blendet eine zuvor eingeblendete Datei für der angegebenen Adressbereich aus.

```
int msync(caddr_t addr, size_t len, int flags)
```

erlaubt in durch `flags` festlegbaren Varianten, den Inhalt der im angegebenen Adressbereich eingeblendeten Datei auf die Originaldatei zurückzuschreiben.

Notizen

Dienste zur Adressraum-Manipulation

In neueren UNIX-Varianten, die gewisse Echtzeitanforderungen erfüllen, wurden auch Funktionen zur Speicherkontrolle bereitgestellt, etwa um sicherzustellen, dass der BS-Kern einem Echtzeitprozess nicht unfreiwillig Seiten verdrängt und dadurch beim nächsten Zugriff (relativ lange) Pagein-Zeiten anfallen.

Operationen

- `memctl(...)`
- `memprotect(...)`
- `mincore(...)`

Hier nicht weiter betrachtet.

Notizen

Zusammenfassung

Hochschule RheinMain

Was haben wir in Kap. 9 gemacht?

Konzepte der Speicherverwaltung (Wichtig!).

Statische Speicherverwaltung als einfachste Verfahren.

Swapping unterstützt Systeme, deren Prozesse mehr Speicher benötigen als insgesamt zur Verfügung steht, lagern dabei aber stets ganze Prozesse aus und ein.

Freispeicherverwaltung im Arbeitsspeicher und auf der Platte geschieht durch Verfahren basierend auf Bitmaps, verketteten Listen oder dem Buddy-System.

Virtueller Speicher unterstützt Systeme, deren Prozesse sehr groß werden können.

Paging und entsprechende Hardware-Unterstützung (Rechnerarchitektur) wurden grundlegend besprochen.

Notizen

Zusammenfassung (2)

Hochschule RheinMain

Seitenersetzungsalgorithmen wurden vielfach untersucht. Clock-Algorithmus und Aging sind praktikabel und brauchbar, der optimale Algorithmus und LRU sind dagegen gut, aber nicht praktikabel.

In konkreten Paging-Systemen gewinnen weitere Aspekte an Bedeutung, etwa die Abwägung des Working Set Modells gegenüber reinem Demand Paging oder lokale gegenüber globaler Seitenersetzung.

Abschließend wurden die UNIX-Systemaufrufe zur Speicherverwaltung besprochen.

Notizen

10. Dateisysteme

https://tagebucheinesdeutschen.wordpress.com/2016/08/06/neues-von-der-scheinbehoerden-front-strategiewechsel-der-firma-finanzamt/comment-page-1/

Notizen

Dateisysteme

Anforderungen an informationsverarbeitende Systeme:

- Speichern und Wiederauffinden sehr großer Mengen von Informationen
- Lebensdauer der Informationen länger als die der benutzenden Prozesse (**Persistenz**).
- Gemeinsame Nutzung von Informationen durch Prozesse (**Sharing**).

Definition: Datei, Dateisystem

Eine **Datei** (*File*) ist eine logische Einheit zur Speicherung von Informationen auf externen Speichermedien.

Dateisysteme (*File Systems*) sind die Teilsysteme eines Betriebssystems, die der Bereitstellung von Dateien dienen

Notizen

Dateisysteme

Hochschule RheinMain

Notizen

Dateien
Verzeichnisse
Implementierung von Dateisystemen
Zusammenfassung

UNIX wird beispielhaft in den jeweiligen Abschnitten behandelt.

Dateien

Hochschule RheinMain

Notizen

Zunächst: Einführung von Dateien aus Benutzersicht

Benennung von Dateien
Dateistrukturen
Dateitypen
Dateizugriff
Dateiattribute
Dateioperationen
Memory-Mapped Files

Benennung von Dateien

Hochschule RheinMain

Datei als Abstraktion zur Speichern und Lesen von Information auf einem Hintergrundspeicher

Benutzer muss **nicht** wissen, wie und wo die Information abgelegt wird, noch wie der Hintergrundspeicher (i.d.R. Platte) im Detail funktioniert.

Dateinamen werden benutzt, um in Dateien abgelegte Information zu identifizieren und wiederaufzufinden:

- Namensvergabe erfolgt bei Dateierzeugung durch den erzeugenden Prozess.
- Datei und Dateiname bleiben bestehen, auch wenn der Prozess terminiert.
- Dateiname kann von anderen Prozessen benutzt werden, um Zugang zu der gespeicherten Information zu bekommen.

Notizen

Benennung von Dateien (2)

Hochschule RheinMain

Definition: Dateinamensraum

Der **Dateinamensraum** definiert die Menge der zulässigen Dateinamen.

Regeln für zulässige Dateinamen sind stark systemabhängig:

Unterscheidung zwischen Groß- und Kleinbuchstaben (ja: UNIX, nein: MS-DOS).

Länge der zulässigen Dateinamen (UNIX: 255 Zeichen, MS-DOS: 8 Zeichen).

Verwendung von Sonderzeichen.[1]

Verwendung von Namenserweiterungen (*File Extensions*):

- optional (Regelfall) oder erzwungen.
- einfach (z.B. MS-DOS) oder mehrfach (UNIX).
- Konventionen für die Verwendung
  - z.B. `.c` für C-Quelldateien
  - z.T. von verarbeitendem Programmen erzwungen z.B. `make`

[1] UNIX: (fast) alles erlaubt, aber besser vermeiden: Leerzeichen, Umlaute...

Notizen

Typische Namenserweiterungen

Hochschule RheinMain

Erweiterung	Bedeutung
.bak	Sicherungsdatei
.BAT	MS-DOS „Batch“ Datei
.c	C-Quelltext
.gif	Bilddatei (Compuserve Graphical Image Format)
.html	Hypertext Markup Language
.jpg, .jpeg	Bilddatei nach dem JPEG Standard
.o	Objektdatei (Übersetzt, noch nicht gebunden)
.mp3	Audiodatei
.pdf	Portable Document Format
.sh	UNIX Shell Script
.tar	UNIX Tape Archive
.tar.gz, .tgz	Komprimiertes UNIX Tape Archive
.tex	Eingabe für das TeX-Satzsystem
.txt	ASCII-Textdatei
.zip	Komprimiertes Archiv

N.B.: UNIX: Lediglich Konvention

Notizen

Dateistrukturen: Bytefolge

Hochschule RheinMain

Die Datei wird als **unstrukturierte Folge von Bytes** aufgefasst.

Das Betriebssystem **weiß nichts** über den Inhalt oder dessen Struktur, sieht eine Datei ausschließlich als Container an.

Interpretation der Bytefolge durch die zugreifenden Anwendungen

Vorteil: Maximale Flexibilität.

Sequenzielle Verarbeitung und/oder wahlfreier Zugriff durch Ansteuern einer Byte-Position

Notizen

Dateistrukturen: Satz-Dateistruktur

Hochschule RheinMain

Satz 0	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6
Meier	Huber	Müller	Nöckel	Scholz	Panik	Hansl
25.1	19.5	11.0	16.2	17.3	18.0	19.2

Dateianfang Satz (Record) Dateiende

Die Datei wird als **Folge von Sätzen fester Länge** aufgefasst.

Das Betriebssystem kennt die Satzlänge, nicht aber die innere Struktur.

Lese / Schreib / Änderungsoperationen stets nur auf ganze Sätze anwendbar.

Sätze sind durch ihre Satznummer (durch „Abzählen“) ansteuerbar.

Ursprung: Dateien entsprechen Lochkartenstapel, d.h. Folge von Sätzen der Länge 80 Zeichen.

Notizen

Dateistrukturen: Baum von Sätzen

Hochschule RheinMain

Die Datei wird als **Baum von Sätzen mit Schlüsselfeld** aufgefasst.

Betriebssystem sieht die eventuell variabel langen Sätze, das Schlüsselfeld an fester Position im Satz und die Ordnung der Sätze.

Operationen:

- Suchen eines Satzes bei gegebenem Schlüsselfeld.
- Ordnungserhaltendes Hinzufügen eines Satzes.

Notizen

Dateitypen

Hochschule RheinMain

UNIX: Zu unterscheiden[2] ...

Gewöhnliche Dateien (*Regular Files*)

- ASCII- („Text-“) Dateien: Folge von Textzeilen variabler Länge, i.d.R. durch **betriebssystemabhängiges** Steuerzeichen getrennt (UNIX: „`\n`“, MacOS: „`\r`“, Windows: „`\r\n`“), mit Texteditor manipulierbar.
- Binärdateien: Alles Andere, verschiedene interne Formate, abhängig von Verwendung, häufig durch sog. „Magic Numbers“ gekennzeichnet, z.B. als ausführbare Dateien.
- (UNIX: **Keine** Unterscheidung auf Dateisystem-Ebene.)

Verzeichnisse (*Directories*) (→ vgl. 10.2)

Zeichenorientierte Spezialdateien (*Character Special Files*)

- Repräsentierung serieller E/A-Geräte, z.B. Terminals (z.B. Linux: „`/dev/ttyS0`“), Drucker, Netzwerk.

Blockorientierte Spezialdateien (*Block Special Files*)

- Repräsentierung blockorientierter Hintergrundspeichermedien, z.B. Platte (z.B. Linux: „`/dev/sda1`“), CDROM.

[2] Tipp: Kommando „`file`“

Notizen

Beispiel: UNIX Executable File

Hochschule RheinMain

Format einer ausführbaren Datei (`a.out`, einfacher als `ELF`):

Notizen

Dateizugriff

Hochschule RheinMain

Sequenzieller Zugriff:

Verarbeitung beginnend am Dateianfang fortlaufend bis zum Dateiende.

„Zurückspulen“ auf den Dateianfang.

→ Angemessene Abstraktion für Dateien auf Magnetbändern.

Direkter Zugriff:

Bytes oder Sätze der Datei können in beliebiger Reihenfolge verarbeitet, gelesen oder geschrieben werden.

Alternativen:

- Jede Operation gibt die Position explizit an.
- Positionierbefehl `seek` innerhalb einer Datei, danach sequenzielle Verarbeitung von dieser Position ausgehend.

In allen neueren Systemen sind Dateien Direktzugriffsdateien.

Notizen

Dateiattribute

Hochschule RheinMain

Definition:

Dateiattribute (auch: Metadaten) sind Eigenschaften, die das Betriebssystem mit einer Datei verbindet (und im Dateikontrollblock speichert).

Menge und Semantik der verwendeten Attribute ist systemabhängig.

Beispiel UNIX:

Dateikontrollblock heißt *inode* (index node) und enthält u.a.:

- Dateityp
- Zugriffsrechte (`rwx` für owner, group, others (vgl. 10.5))
- Anzahl Referenzen auf diese Datei (Reference Counter)
- Eigentümer (user id und group id)
- Größe (Anzahl bytes)
- Zeitpunkt des letzten Zugriffs
- Zeitpunkt der letzten Modifikation
- Zeitpunkt des letzten Updates dieses Kontrollblocks

In UNIX wird der Dateiname selbst nicht im inode gespeichert, sondern im entsprechenden Verzeichnis (vgl. 10.2, 10.3).

Notizen

Mögliche Dateiattribute

Hochschule RheinMain

Attribute	Bedeutung
Schutz	Wer kann wie auf Datei zugreifen?
Passwort	Passwort für den Zugriff
Urheber	ID der Person, die die Datei erzeugt hat
Eigentümer	Aktueller Eigentümer
Read-only-Flag	0: Lesen/Schreiben, 1: nur Lesen
Hidden-Flag	0: normal, 1: in Listen nicht sichtbar
System-Flag	0: normale Datei, 1: Systemdatei
Archiv-Flag	0: wurde gesichert, 1: muss noch gesichert werden
ASCII/Binär-Flag	0: ASCII-Datei, 1: Binärdatei
Random-Access-Flag	0: nur sequenzieller Zugriff, 1: wahlfreier Zugriff
Temporary Flag	0: normal, 1: Datei bei Prozessende löschen
Sperr-Flags	0: nicht gesperrt, nicht null: gesperrt
Datensatzlänge	Anzahl der Bytes in einem Datensatz
Schlüsselposition	Offset des Schlüssels innerhalb des Datensatzes
Schlüssellänge	Anzahl der Bytes im Schlüsselfeld
Erstellungszeit	Datum und Zeitpunkt der Dateierstellung
Zeitpunkt des letzten Zugriffs	Datum und Zeitpunkt des letzten Zugriffs
Zeitpunkt der letzten Änderung	Datum und Zeitpunkt der letzten Dateiänderung
Aktuelle Größe	Anzahl der Bytes in der Datei
Maximale Größe	Anzahl der Bytes für maximale Größe der Datei

Notizen

Dateioperationen

Hochschule RheinMain

Typische Dateioperationen

- `Create` – (Leere) Datei erzeugen
- `Open` – Datei vor Benutzung öffnen
- `Append` – Daten am Ende der Datei anfügen
- `Delete` – Datei löschen
- `Close` – Datei nach Benutzung schließen (noch ausstehende Schreibvorgänge ausführen)
- `Read` – Daten ab aktueller Position lesen
- `Write` – Daten ab aktueller Position schreiben
- `Seek` – Aktuelle Position setzen
- `Get Attributes` – Dateiattribute lesen
- `Set Attributes` – Dateiattribute setzen
- `Rename` – Dateinamen ändern

Notizen

UNIX: Create

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

Oder (äquivalent):

```
int open(const char *path, O_CREAT|O_WRONLY|O_TRUNC, mode_t mode);
```

Erzeugt neue Datei mit Namen `path` und liefert einen Dateideskriptor als Ergebnis (-1 bei Fehler). Symbolische Werte für `mode`:

Symbol	Oktalzahl	Bedeutung
S_IRUSR	0400	Benutzer hat Leserechte
S_IWUSR	0200	Benutzer hat Schreibrechte
S_IXUSR	0100	Benutzer hat Ausführrechte
S_IRWXG	0070	Gruppe hat Lese-, Schreib- und Ausführrechte
S_IRGRP	0040	Gruppe hat Leserechte
S_IWGRP	0020	Gruppe hat Schreibrechte
S_IXGRP	0010	Gruppe hat Ausführrechte
S_IRWXO	0070	andere haben Lese-, Schreib- und Ausführrechte
S_IROTH	0004	andere haben Leserechte
S_IWOTH	0002	andere haben Schreibrechte
S_IXOTH	0001	andere haben Ausführrechte

`mode` wird mit der Prozess-Dateierzeugungsmaske `umask` zur Festlegung der Zugriffsrechte so verknüpft, dass Zugriffsrechte für gesetzte `umask`-Bits weggenommen werden.

Notizen

UNIX: Open, Append

```
#include <fcntl.h>

int open(const char *path, int flags, mode_t mode);
int open(const char *path, int flags);
```

Öffnet Datei mit Namen `path` und liefert einen Dateideskriptor als Ergebnis (-1 bei Fehler). `flags` – Wie soll die Datei geöffnet werden? (→ `fcntl.h`). Mögliche Werte:

`O_RDONLY` = nur lesen
`O_WRONLY` = nur schreiben
`O_RDWR` = lesen und schreiben

Per Bit-ODER („|“) können verschiedene Flags hinzugefügt werden:

Flag	Bedeutung
O_APPEND	Schreibzugriffe: am Dateiende anhängen
O_CREAT	Datei anlegen, falls noch nicht vorhanden (in diesem Fall Argument `mode` erforderlich)
O_EXCL	(mit `O_CREAT`) Fehler, falls Datei schon da
O_TRUNC	löscht Dateiinhalt, falls schon vorhanden
O_NDELAY	erlaubt nicht-blockierende E/A-Operationen
O_SYNC	sofortiges Zurückschreiben bei Schreiboperationen

Behandlung von Argument `mode`: s.o. (`creat()`)

Notizen

UNIX: Delete, Close

```
#include <unistd.h>

int unlink(const char *path);
```

Löscht den Namen `path` aus dem Dateisystem. Falls dieser Name der letzte Link auf eine Datei war und kein Prozess die Datei geöffnet hält, wird sie gelöscht.

```
#include <unistd.h>

int close(int fd);
```

Schließt den Dateideskriptor `fd`, so dass dieser nicht mehr zu einer Datei gehört und wieder verwendet werden kann.

Notizen

UNIX: Read, Write

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Liest **bis zu** `count` Byte aus dem Dateideskriptor `fd` in den bei `buf` beginnenden Puffer. Rückgabewert: Anzahl **tatsächlich gelesener** Bytes (0 bedeutet Dateieinde), oder -1 bei Fehler.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Schreibt **bis zu** `count` Byte aus dem bei `buf` beginnenden Puffer in die Datei, auf die der Dateideskriptor `fd` weist. Rückgabewert: Anzahl **tatsächlich geschriebener** Bytes, oder -1 bei Fehler.

Notizen

UNIX: Seek

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

`lseek()` positioniert die aktuelle Dateiposition von `fd` auf den Wert `offset` gemäß der Einstellung von `whence`. `offset` darf auch negativ sein. Mögliche Werte für `whence`:

Symbol	Bedeutung
SEEK_SET	neue Position wird auf offset gesetzt
SEEK_CUR	neue Position ist aktuelle Position + offset
SEEK_END	neue Position ist Dateiende + offset

Rückgabewert: neue Position (ab Anfang gezählt) oder -1 bei Fehler

Notizen

UNIX: Get Attributes

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

Liefert Informationen zu der durch `path` angegebenen Datei in der angegebenen `stat`-Struktur (-1 bei Fehler). Identisch: `fstat` bei bereits geöffneter Datei.

```
struct stat {
    dev_t     st_dev;      /* Geraet */
    ino_t     st_ino;      /* Inode */
    mode_t    st_mode;     /* Dateityp und -modus */
    nlink_t   st_nlink;    /* Anzahl harter Links */
    uid_t     st_uid;      /* UID des Besitzers */
    gid_t     st_gid;      /* GID des Besitzers */
    dev_t     st_rdev;     /* Typ (wenn Inode-Geraet) */
    off_t     st_size;     /* Groesse in Bytes */
    blksize_t st_blksize;  /* Blockgroesse fuer Dateisystem-E/A */
    blkcnt_t  st_blocks;   /* Anzahl der zugewiesenen 512B-Bloecke */
    time_t    st_atime;    /* Zeit des letzten Zugriffs */
    time_t    st_mtime;    /* Zeit der letzten Veraenderung*/
    time_t    st_ctime;    /* Zeit der letzten Statusaenderung*/
};
```


Notizen

UNIX: Set Attributes

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Ändert den Modus der angegebenen Datei, deren Pfadname in `path` übergeben wird, zu `mode`. Identisch: `fchmod` bei bereits geöffneter Datei.

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Ändert den Eigentümer und Gruppenzugehörigkeit der angegebenen Datei, deren Pfadname in `path` übergeben wird. Identisch: `fchown` bei bereits geöffneter Datei. ID-Nummern für Eigentümer (`uid`) und Gruppe (`gid`) stehen z.B. in der Datei `/etc/passwd`; Angabe von -1: keine Änderung.

```
#include  <sys/types.h>
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *zeiten);
```

Zugriffs- und Modifikationszeitpunkte setzen.

Notizen

UNIX: Rename

```
#include  <stdio.h>

int rename(const char *oldpath, const char *newpath);
```

Benennt eine Datei um und verschiebt sie in ein anderes Verzeichnis, wenn nötig. Alle anderen Hard Links sind nicht betroffen, ebenso offene Dateideskriptoren für `oldpath`.

Notizen

Memory-Mapped Files

Hochschule RheinMain

Definition: Memory Mapping

Memory-Mapping von Dateien bezeichnet das Einblenden und Ausblenden von Dateien (oder Teil-Fenstern) in den Adressraum eines Prozesses.

Abstrakte Operationen: `Map` und `Unmap`

Vorteil: Zugriff auf Dateien mit normalen Speicherbefehlen, kein `read()` und `write()` erforderlich.
In neueren Systemen in Zusammenhang mit Virtual Memory Management zum Transport der Seiten der Datei realisiert.

Notizen

UNIX: Map, Unmap

Hochschule RheinMain

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
                int fd, off_t offset);
```

Blendet `length` Bytes von Datei mit Dateideskriptor `fd` ab Position `offset` in den Adressraum des aktuellen Prozesses ab Adresse `addr` ein. Falls `addr == NULL` → System wählt Adresse selbst.

`prot` = Zugriffsart
mögliche Werte:

Flag	Bedeutung
PROT_EXEC	Ausführen
PROT_READ	Lesen
PROT_WRITE	Schreiben
PROT_NONE	Kein Zugriff

`flags`: weitere Optionen
Auswahl möglicher Werte:

Flag	Bedeutung
MAP_SHARED	Änderungen für andere Prozesse sichtbar
MAP_PRIVATE	Änderungen nicht sichtbar
MAP_ANONYMOUS	Keine Hintergrunddatei
MAP_FIXED	Keine Adressauswahl durch das BS
…	…

Rückgabewert: Virtuelle Adresse des Bereiches, (void*)-1 bei Fehler.

```
int munmap(void *addr, size_t length);
```

Aufheben des Mappings

Notizen

Beispiel: Direktzugriff

Hochschule RheinMain

`struct`-Typ beschreibt Messwerte-Datensatz:

```
typedef struct mw {
    char ableser[20];
    float temperatur;
} Messwert;
```

Ziel:

Speichern der Messwerte der jeweils letzten 7 Tage (rollierend)
Dateiposition aus Wochentag (0=Sonntag, 1=Montag, ...)
Direkter Zugriff über Wochentag-Nummer

```
enum { SONNTAG, MONTAG, .. SAMSTAG } wt;
```

Satz 0	Satz 1	Satz 2	Satz 3	Satz 4	Satz 5	Satz 6
Meier	Huber	Müller	Nöckel	Scholz	Panik	Hansl
25.1	19.5	11.0	16.2	17.3	18.0	19.2

Notizen

Beispiel: Direktzugriff (2)

Hochschule RheinMain

```
int speichern(int fd, Messwert *pm, int tag) {
    if(lseek(fd, tag*sizeof(Messwert), SEEK_SET) < 0) {
        perror("speichern (lseek)");
        return -1;
    }
    if(write(fd, pm, sizeof(Messwert)) < 0) {
        perror("speichern (write)");
        return -1;
    }
    return 0;
}
```

```
main() {
    Messwert m;
    enum { SONNTAG, MONTAG, DIENSTAG, MITTWOCH, ...};
    ...
    fd = open("messwerte.dat", O_RDWR | O_CREAT, 0644);
    err1 = speichern(fd, &m, SONNTAG);
    err2 = lesen(fd, &m, MONTAG); /* wie speichern() */
    ...
}
```


Notizen

Beispiel: Mmap

```
...
int main(void) {
    int fd, laenge, i;
    Messwert *pmw; /* s.o.: "Direktzugriff" */
    fd = open("messwerte.dat", O_RDWR, 0644);
    laenge = lseek(fd, 0, SEEK_END);

    pmw = (Messwert*)mmap(0, laenge, PROT_READ|PROT_WRITE,
                          MAP_SHARED, fd, 0);
    if((Messwert*)-1 == pmw) {
        perror("mmap()");
        exit(EXIT_FAILURE);
    }
    for(i=0; i < 3; i++) {
        printf("Ableser %s: %f Grad\n", pmw[i].ableser,
                                        pmw[i].temperatur);
        pmw[i].temperatur = pmw[i].temperatur * 2;
    }
    munmap(pmw, laenge);
    return 0;
}
```


Notizen

Verzeichnisse

Definition: Verzeichnis

Ein **Verzeichnis** (Ordner, Directory) dient der Strukturierung der Dateimenge eines Dateisystems und besteht aus einer Menge von Einträgen für Dateien und i.d.R. für weitere Verzeichnisse (Unterverzeichnisse).

- Verzeichnissysteme mit einer Ebene
- Hierarchische Verzeichnissysteme
- Pfadnamen
- Verzeichnisoperationen
- Montierbare Verzeichnisbäume

Notizen

Verzeichnissysteme mit einer Ebene

Hochschule RheinMain

Einfachste Form:

Alle Dateien in einem (Wurzel-)Verzeichnis

Varianten:

- Ein Verzeichnis pro Laufwerk
- Ein Verzeichnis pro Benutzer:In

Früher in Single User Systemen (PCs) gebräuchlich

- CP/M, erste MS-DOS Versionen

Vorteil: Einfach

Nachteil: Nur bei überschaubarer Anzahl Dateien brauchbar

Notizen

Hierarchische Verzeichnissysteme

Hochschule RheinMain

Verzeichnisbaum für jeden Benutzer zur Strukturierung seiner Dateimenge (heute Standard).

Darüber hinaus Einführung zusätzlicher **symbolischer Links** zur flexiblen gemeinsamen Benutzung von Dateimengen.
Dateisystemstruktur wird damit zu einem **azyklischen Graphen**.

Notizen

Pfadnamen

Hochschule RheinMain

Definition: Pfadnamen

Für hierarchische Verzeichnissysteme dienen **Pfadnamen** (*Path Names*) der Benennung von Dateien und Verzeichnissen.

Alternativen

Absolute Pfadnamen:

- UNIX: „/“ Name des Wurzelverzeichnisses
- Benennung vom Wurzelverzeichnis aus.
- UNIX: `/usr/egon/uebung/mist`.
- Beispiel MS-DOS: „\“ als Separator.

Relative Pfadnamen:

- Namen werden von einem Arbeitsverzeichnis ausgehend interpretiert (akt. Verzeichnis, Working oder Current Directory).
- „`.`“ bezeichnet häufig das aktuelle Verzeichnis selbst.
- „`..`“ bezeichnet häufig das Eltern-Verzeichnis (*Parent Directory*).

Notizen

Typische System-Verzeichnisstrukturen

Hochschule RheinMain

https://de.wikipedia.org/w/index.php?curid=4098994

Notizen

Linux Filesystem Hierarchy Standard (FHS)

Hochschule RheinMain

/ bin	(binaries) wichtige Dienstprogramme, bereits für Hochfahren
sbin	wichtige Dienstprogramme für Systemverwalter, (single user)
dev	(devices) Gerätedateien
etc	Systemverwaltungsprogramme und Systemdateien
home	Heimatverzeichnisse der Benutzer
media	(mount) für temporär einhängbare Dateisysteme (s.u.)
opt	(optional) optionale Software-Pakete
proc	(process) Pseudoverzeichnis des Prozessdateisystems
tmp	für temporäre Dateien von Anwendungen
usr	(user) wichtiges Verz. (Dateisystem), in Netzen häufig read-only mountbar
usr/bin	Dienst- und Anwendungsprogramme
usr/include	Header-Dateien
usr/lib	Bibliotheken, Programme, ... (Sammelverzeichnis)
usr/local	Komponenten, die spezifisch für die Installation sind
usr/share	nur einmal zu speichernde Information mehrerer Varianten
var	ausgegliederter, veränderlicher Teil des usr-Verzeichnisses

Notizen

Verzeichnisoperationen

Hochschule RheinMain

Typische Verzeichnisoperationen

- `Create` – Verzeichnis erzeugen
- `Delete` – Verzeichnis löschen
- `Changedir` – Verzeichnis wechseln
- `Opendir` – Verzeichnis zum Lesen vin Verzeichniseinträgen öffnen
- `Closedir` – Verzeichnis nach Benutzung schließen
- `Readdir` – Nächsten Verzeichniseintrag lesen
- `Rename` – Verzeichnis umbenennen
- `Link` – Verzeichniseintrag für bestehende Datei erzeugen
- `Unlink` – Verzeichniseintrag (und ggf Datei) löschen

Notizen

UNIX: Create, Delete, Changedir

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int mkdir(char *path, mode_t mode);
int rmdir(char *path);
int chdir(char *path);
```

`mkdir()` legt das Verzeichnis `path` mit Zugriffsrechten `mode` und Einträgen für „`.`“ und „`..`“ an.
`rmdir()` löscht das (bis auf die Einträge „`.`“ und „`..`“ leere!) Verzeichnis `path`
`chdir()` setzt das aktuelle Verzeichnis **für den ausführenden Prozess** auf `path`

Wieso ist der Verweis-Zähler für ein Verzeichnis mindestens 2?

```
$ mkdir beispiel
$ ls -l
drwx------ 2 kaiser profs 4096 Apr 20 15:00 beispiel
```


Notizen

UNIX: Opendir, Closedir, Readdir

```
#include <dirent.h>
#include <sys/types.h>

DIR *opendir(char *path);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dir);
```

`opendir()` öffnet das Verzeichnis `path` und gibt einen Zeiger auf `DIR` zurück (`NULL` bei Fehler)
`closedir()` schließt ein Verzeichnis; Ergebnis ist 0 (ok) oder -1 (Fehler)
`readdir()` liefert den jeweils nächsten Verzeichniseintrag. Bei Ende oder Fehler wird der `NULL`-Zeiger geliefert.

`struct dirent` enthält ein Feld `char d_name[]` mit dem Namen des betreffenden Verzeichniseintrags:

```
struct dirent {
  ino_t          d_ino;       /* Inode-Nummer */
  off_t          d_off;       /* besser nicht verwenden, (s. manpage) */
  unsigned short d_reclen;    /* Laenge dieses Datensatzes */
  unsigned char  d_type;      /* Dateityp; nicht immer unterstuetzt */
  char           d_name[256]; /* Null-terminierter Dateiname */
};
```


Notizen

Beispiel: mini-ls

Hochschule RheinMain

Ziel: So etwas ...

```
$ ./a.out /etc
[/etc/sysconfig]
[/etc/X11]
/etc/fstab (1355 Bytes)
/etc/mtab (413 Bytes)
/etc/modules.conf (1049 Bytes)
/etc/csh.cshrc (561 Bytes)
/etc/bashrc (1497 Bytes)
/etc/gnome-vfs-mime-magic (8042 Bytes)
[/etc/profile.d]
/etc/csh.login (409 Bytes)
/etc/exports (2 Bytes)
/etc/filesystems (51 Bytes)
/etc/group (601 Bytes)
/etc/host.conf (17 Bytes)
/etc/hosts.allow (161 Bytes)
/etc/hosts.deny (347 Bytes)
...
```


Notizen

Beispiel: mini-ls (2)

Hochschule RheinMain

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  DIR *dir;
  struct dirent *eintrag;
  struct stat statbuf;
  char pfadpuffer[PATH_MAX], *pfadp;

  if (argc != 2) {
    printf("Aufruf: %s verzeichnis\n",argv[0]);
    exit(EXIT_FAILURE);
  }
  dir = opendir(argv[1]);
  if (dir == NULL) {
    perror(argv[1]);
    exit(EXIT_FAILURE);
  }
```


Notizen

Beispiel: mini-ls (3)

Hochschule RheinMain

```
    strcpy(pfadpuffer, argv[1]);
    strcat(pfadpuffer, "/");
    pfadp = pfadpuffer + strlen(pfadpuffer);
    while (1) {
        eintrag = readdir(dir);
        if (eintrag == NULL) break;
        if (strcmp(eintrag->d_name, ".")==0 ||
            strcmp(eintrag->d_name, "..")==0) continue;
            strcpy(pfadp, eintrag->d_name);

        if (stat(pfadpuffer, &statbuf) == -1) {
            perror("stat()");
        } else if (S_ISDIR(statbuf.st_mode)) {
            printf("[%s]\n",pfadpuffer);
        } else {
            printf("%s (%ld Bytes)\n",pfadpuffer, statbuf.st_size);
        }
    }
    closedir(dir);
    return EXIT_SUCCESS;
}
```

S_ISDIR(m)=TRUE

⇒ Ist Verzeichnis

Notizen

UNIX: Rename, Link, Unlink

Hochschule RheinMain

```
#include  <stdio.h>

int rename(const char *oldpath, const char *newpath);

#include <unistd.h>

int link(char *oldpath, char *newpath);
int unlink(char *path);
```

`rename()` Benennt ein Verzeichnis um und verschiebt es in ein anderes Verzeichnis, wenn nötig.

- kein Unterschied zu Dateioperation `rename()` (s.o.)
- entsprechendes shell-Kommando: `mv oldpath newpath`

`link()` legt einen neuen Verzeichniseintrag `newpath` an, der auf dieselbe Datei („inode“) wie der bestehende `oldpath` verweist, und erhöht den Referenzzähler im inode. (**Keine** Datei-Kopie!)

- entsprechendes shell-Kommando: `ln oldpath newpath`

`unlink()` erniedrigt den Referenzzähler im zugehörigen inode und löscht den Verzeichniseintrag, sowie die Datei, falls der Referenzzähler auf 0 gefallen ist.

- kein Unterschied zu Dateioperation `unlink()` (s.o.)
- entsprechendes shell-Kommando: `rm path`

Notizen

Beispiel: ls -li

`ls -l` zeigt Anzahl der Verweise (Links) auf den inode einer Datei:

```
$ ls -l /usr/bin/
...
-rwxr-xr-x 2 root root 183136 Jul 30 2019 unzip
...
-rwxr-xr-x 3 root root 7584 Jan 6 2019 zegrep
-rwxr-xr-x 3 root root 7584 Jan 6 2019 zfgrep
-rwxr-xr-x 1 root root 2080 Jan 6 2019 zforce
-rwxr-xr-x 3 root root 7584 Jan 6 2019 zgrep
-rwxr-xr-x 1 root root 213136 Aug 16 2015 zip
-rwxr-xr-x 2 root root 183136 Jul 30 2019 zipinfo
```

Option „`-i`“ zeigt zusätzlich die inode-Nummer → so werden verschiedene Verweise auf denselben inode erkennbar:

```
$ ls -li /usr/bin/z*grep
132224 -rwxr-xr-x 1 root root 7584 Jan 6 2019 /usr/bin/zegrep
132224 -rwxr-xr-x 1 root root 7584 Jan 6 2019 /usr/bin/zfgrep
132224 -rwxr-xr-x 1 root root 7584 Jan 6 2019 /usr/bin/zgrep
155628 -rwxr-xr-x 1 root root 2953 Jul 30 2019 /usr/bin/zipgrep
```

→ `zegrep`, `zfgrep` und `zegrep` sind hier **dieselbe Datei**
→ Unterscheidung der Funktion anhand `argv[0]`

Notizen

UNIX: Symbolische Links

```
#include <unistd.h>

int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
```

`symlink()` legt einen **symbolischen** Verweis `newpath` an, der auf `oldpath` verweist.

- entsprechendes shell-Kommando: `ln -s oldpath newpath`

`readlink()` liest den Verweis aus Symlink `path` in den Zeichenvektor `buf` (max. `bufsiz` Zeichen).
(Für beide gilt: Ergebnis 0 für „ok“, -1 für Fehler.)

Unterschied zu bisher betracheten *Hard Links*:

Verweis erfolgt **per Namen**, nicht inode (ändert Referenzzähler nicht).
Konsequenzen:

- Auflösung zur Laufzeit nötig, evtl. mehrstufig
- Verweise über Filesystem- und Partitionsgrenzen hinweg möglich (warum geht das mit harten Links nicht?)
- Symbolische Links können „ins Leere“ verweisen (Ziel existiert nicht)

Notizen

Beispiel: Symbolische Links: ls -li

```
$ ls -l /usr/lib/cpp
lrwxrwxrwx 1 root root 21 Mai 5 2020 /usr/lib/cpp -> /etc/alternatives/cpp

$ ls -l /etc/alternatives/cpp
lrwxrwxrwx 1 root root 12 Mai 5 2020 /etc/alternatives/cpp -> /usr/bin/cpp

$ ls -l /usr/bin/cpp
lrwxrwxrwx 1 root root 5 Feb 25 2019 /usr/bin/cpp -> cpp-8

$ ls -l /usr/bin/cpp-8
lrwxrwxrwx 1 root root 22 Apr 6 2019 /usr/bin/cpp-8 -> x86_64-linux-gnu-cpp-8

$ ls -l /usr/bin/x86_64-linux-gnu-cpp-8
-rwxr-xr-x 1 root root 1104760 Apr 6 2019 /usr/bin/x86_64-linux-gnu-cpp-8
```

Verweisziel wird bei symbolischen Links von `ls` angezeigt („->“)
Typkennzeichen in `ls`-Ausgabe: „`l`“ (symLink)
Hier: Zugriff auf `/usr/lib/cpp` führt *letztlich* auf
`/usr/bin/x86_64-linux-gnu-cpp-8`

Notizen

Montierbare Verzeichnisbäume

Der sichtbare Dateiraum ist i.d.R. durch mehrere Dateisysteme auf mehreren Speichern (u.U. auf mehreren Rechnern) realisiert.

⇒ Mehr oder weniger sichtbare Auswirkungen auf den Dateinamensraum.

Alternativen:

- Dateinamen spiegeln die verschiedenen Speicher wider.
  - ★ Beispiel: Laufwerksbuchstaben in Windows: `C:\WINDOWS\system`, `H:\setup.exe`
- Transparenz im Dateinamensraum.
  - ★ Der gesamte Dateiraum ist aus mehreren Dateisystemen mit jeweils eigenem Verzeichnisbaum durch **Montieren** (*mount*) zusammengesetzt.
  - ★ Nach dem Montieren existiert ein einziger Dateinamensbaum.
  - ★ Beispiel: UNIX, Kommandos: `mount` / `umount`

Notizen

Beispiel:

Notizen

Beispiel: /etc/fstab

zu mountendes Gerät
Mountpunkt
Dateisystem-Typ
Mount-Optionen
Informationen für Datensicherung (dump)
Rangfolge bei der Konsistenzprüfung

```
# <file system> <mount point> <type>      <options>       <dump/pass>
/dev/sda1       /             ext4        errors=remount-ro 0   1
/dev/sda2       /var          ext4        defaults          0   2
/dev/sda3       none          swap        sw                0   0
/dev/sdb1       /home         ext4        defaults          0   2
/dev/sdb2       /opt          ext4        defaults          0   2
/dev/sr0        /media/cdrom0 udf,iso9660 user,noauto       0   0
```

Notizen

Implementierung von Dateisystemen

Hochschule RheinMain

Im folgenden Sichtweise der Implementierung

- Technische Gegebenheiten
- Plattenplatz-Verwaltung
- Implementierung von Dateien
- Implementierung von Verzeichnissen
- Dateisystemstruktur
- Plattenplatz-Kontingentierung
- Wahl der Blockgröße
- Repräsentierung im Hauptspeicher
- Zuverlässigkeit des Dateisystems
- Performance des Dateisystems

Notizen

Speichermedium: Magnetplatte

Hochschule RheinMain

- 5000 bis über 10000 Umdrehungen/Minute
- Datenübertragungsraten: mehrere hundert Mbit/s
- Mittlere Positionierungszeit: ca. 6ms und weniger

Notizen

Beispiel Festplatte: Technische Daten[3]

Hochschule RheinMain

Hard Disk Model	WDC WD5000AAKX
Disk Family	Caviar Blue
Form Factor	3.5“
Capacity	500 GB (500 x 1 000 000 000 bytes)
Number Of Disks	1
Number Of Heads	2
Rotational Speed	7200 RPM
Rotation Time	8.33 ms
Average Rotational Latency	4.17 ms
Disk Interface	Serial-ATA/600
Buffer-Host Max. Rate	600 MB/seconds
Buffer Size	16384 KB
Average Seek Time	8.9 ms
Track To Track Seek Time	2 ms
Full Stroke Seek Time	21 ms

Maßgeblich für Roh-Datenrate

Spitzen-Datenrate (Schnittstellen-Eigenschaft)

Bedingt durch Kopfbewegung → Spitzenrate wird nicht dauerhaft erreicht

[3] Quelle: `https://www.hdsentinel.com/storageinfo_details.php?lang=en&model=WDC%20WD5000AAKX`

Notizen

Beispiel Festplatte: Geometrie

Hochschule RheinMain

Sektor: Kleinste adressierbare Einheit („Block“)

Zylinder: Spuren mit gleichem Radius
→ Ohne mechanische Kopfbewegung erreichbar.

Blockadressierung durch Zylinder-, Kopf- und Sektornummer (CHS)

Notizen

Eigenschaften von Festplatten

Hochschule RheinMain

Beobachtungen

Wahlfreiheit gilt nur bis zur Ebene der Blöcke (kleinste adressierbare Einheit), die Anwenderschnittstelle (s.o.) ermöglicht jedoch Byte-Adressierung
Die Änderung eines einzelnen Bytes auf der Festplatte erfordert Lesen, Ändern und Zurückschreiben eines ganzen Blockes.
Heutige Festplatten verwenden logische Blockadressen (LBA), die ursprünglich aus Zylinder-, Kopf- und Sektornummer gebildet wurden:

$$LBA = (C \cdot N_{Heads} + H) \cdot N_{BlocksPerTrack} + S$$

Hierdurch können auch Geometriedaten der Festplatte, Eigenschaften wie z.B. eine variable Anzahl $N_{BlocksPerTrack}$, etc. abstrahiert werden → bei heutigen Festplatten haben diese Parameter -sofern sie überhaupt angegeben werden- keinen Bezug zur Realität mehr
Dennoch gilt, dass benachbarte Blöcke mit hoher Wahrscheinlichkeit ohne Kopfbewegungen „in einem Zug“ schneller gelesen / geschrieben werden können.

Notizen

Speichermedium: Halbleiterspeicher

Hochschule RheinMain

Transistor

Speichertechnologie: NAND-Flash, Speicherung von Ladungen auf Transistor-Gates. Typen:

NAND-Typ	Anzahl Zustände	Bits pro Transistor	Lebensdauer (Schreibzyklen)
SLC	1	1	~100.000
MLC	4	2	~5.000
TLC	8	3	~1.000
QLC	16	4	~1.000

Blockgröße: 4KB (Lesen), >256KB (Löschen)
Keine mechanischen Bewegungen notwendig → schnelles Lesen
Änderung eines Bytes erfordert Lesen, Ändern, Löschen und Zurückschreiben eines Blockes
Begrenzte Lebensdauer: Verschleiß durch Löschen
→ *Wear Leveling* zur Haltbarkeitssteigerung

Notizen

(Platten-)platzverwaltung

Hochschule RheinMain

Hauptgesichtspunkte:

Partitionierung.

Logische Laufwerke.

Notizen

Partitionierung

Hochschule RheinMain

In der Regel wird der gesamte Platz auf mehrere *Partitionen* verteilt

MBR *master boot record*: enthält ausführbaren Code, der beim Systemstart vom BIOS (basic input/output system) geladen und gestartet wird.
Darin enthalten: **Partitionstabelle**: beschreibt die Aufteilung der Platte in bis zu 4 Partitionen (Anfang, Länge, Typ, ggf. „bootbar"-Flag)
Der MBR-Code identifiziert eine **Startpartition**, lädt und startet deren ersten Block (**Bootblock**), der seinerseits ggf. das Laden und Starten des Betriebssystem auslöst.
Der Bootblock muß das Dateisystem des zu startenden Betriebssystems (zumindest eingeschränkt) verstehen, der Code im MBR kann unabhängig davon sein.

Notizen

Logische Laufwerke

Hochschule RheinMain

Definition: Logisches Laufwerk

Ein **logisches Laufwerk** (*logical volume*) ist eine dynamisch veränderbare Partition, die sich auch über mehrere physische Datenträger hinweg erstrecken kann.

Beispiel: Linux LVM

Physische Speichergeräte (physical volumes) werden zu Laufwerksgruppen (volume groups) zusammengefaßt
Auf einer Laufwerksgruppe können logische Laufwerke eingerichtet (entspricht Partitionierung) und mit einem Dateisystem versehen werden.

Notizen

LVM: Nutzen

Hochschule RheinMain

Im laufenden Betrieb (!) ...

kann die Kapazität der Laufwerksgruppe durch **Hinzufügen weiterer physische Volumes** vergrößert werden.

können **Daten** von alten Laufwerken auf neue **verlagert** und die alten Laufwerke außer Betrieb genommen werden.

kann logischen Laufwerken mehr **Speicherplatz zugeordnet** werden oder Speicherplatz **entzogen** werden.

LVM unterstützt „Filesystem Snapshots“

Beim Anlegen eines Snapshots wird ein neues logisches Laufwerk angelegt, das den **momentanen Zustand** seines zugehörigen Ursprungs-Laufwerks enthält (eingefrorene Sicht, keine Kopie)

→ Ermöglicht **konsistente Backups** über Snapshot-Laufwerk **trotz weiterlaufenden Betriebs** auf dem ursprünglichen Laufwerk

Notizen

RAID

Hochschule RheinMain

Weitere Möglichkeit: RAID:
Redundant Array of Independent (Inexpensive) Disks

Mehrere Platten zusammengeschaltet, sehen für den Rechner wie **eine große Platte** aus.

Realisierungen:

- Hardware-RAID (spezieller Festplatten-Controller)
- Software-RAID (Betriebssystem verwaltet mehrere angeschlossenen Platten als RAID, Beispiel: Linux `mdadm`)

Ziele:

- **Erhöhung der Datensicherheit** durch geschickte redundante Speicherung.
- Austausch defekter Platten im laufenden Betrieb ohne Unterbrechung (oft auch "hot standby"-Platte)
- Verteilung der Daten auf die einzelnen Platten wird durch „RAID level" (RAID level 0 ... RAID level 6) definiert.

Notizen

RAID0 - „striping"

Hochschule RheinMain

RAID-Platte wird in „Streifen" mit k Blöcken eingeteilt

Streifen werden reihum auf den angeschlossenen Platten abgelegt.

→ **keine Redundanz**, damit keine höhere Fehlertoleranz

→ **schneller Zugriff** besonders bei großen Dateien, da Platten parallel arbeiten können

RAID-Kapazität: Summe der Plattenkapazitäten

Notizen

RAID1 - „mirrorring“

Zu jeder Platte gibt es eine Spiegelplatte gleichen Inhalts

Fehlertoleranz: Wenn eine Platte ausfällt, kann andere sofort einspringen (übernimmt Controller automatisch)

Schreiben: etwas langsamer; **Lesen**: schneller durch Parallelzugriff auf beide zuständigen Platten

RAID-Kapazität: Hälfte der Summe der Plattenkapazitäten

Notizen

RAID5 - parity

Paritätsinformation (XOR) auf alle Platten verteilt
Beispiel: P 0-2 enthält XOR-Verknüpfung über die Streifen 0, 1, 2
XOR Verknüpfung ist „selbstinvers“:

Wenn gilt:	$P = A \oplus B \oplus C$
dann ist:	$A = B \oplus C \oplus P$
und:	$B = A \oplus C \oplus P$
und:	$C = A \oplus B \oplus P$

+ Fehlertolerant bei guter Kapazitätsnutzung
+ Leseoperationen schnell
- Schreiben aufwändiger

→ Solange maximal eine (beliebige) Platte ausfällt, kann ihr Inhalt aus den Übrigen (im lfd. Betrieb) rekonstruiert werden (wieder per XOR)

Notizen

Aufgaben des Dateisystems

Hochschule RheinMain

Wollen Benutzende Blöcke, Sektoren, Spuren etc. selbst ansteuern, ihre Daten in 512-Byte-Blöcke aufteilen müssen usw? Wohl kaum. Sie wollen:

- Mithilfe der in 10.1 beschriebenen Funktionen Dateien und Verzeichnisse bearbeiten und verwalten
- Optimale Hardwarenutzung[4]
- **Einheitlichen Zugang** zu vielen (verschiedenen) Speichergerätearten (standardisierte Schnittstelle)

[4] (natürlich ohne eigene Hardware-Kenntnisse ...)

Notizen

Implementierung von Dateien

Hochschule RheinMain

Hauptproblem

Verwaltung der Plattenblöcke und ihrer Zugehörigkeit zu einer Datei.

Alternativen:

- Kontinuierliche Allokation.
- Allokation mittels einer verketteten Liste.
- Allokation mittels einer verketteten Liste und einem Index.
- Allokation mittels Index Nodes (*inodes*).

Notizen

Kontinuierliche Allokation

Jeder Datei wird eine Menge zusammenhängender Blöcke zugeordnet (beim **Anlegen reserviert**).

Vorteile:

- Einfach zu implementieren (nur die Adresse des ersten Blocks ist zu speichern).
- Sehr gute Performance beim Lesen und Schreiben der Datei (minimale Kopfbewegungen).

Nachteile:

- Maximalgröße der Datei muss zum Erzeugungszeitpunkt bekannt sein, Wachsen (Append) ist nicht möglich
- Externe Fragmentierung durch Löschen / Überschreiben
- Verdichtung extrem aufwändig / langwierig

Einsatzgebiete:

- Echtzeit-Anwendungen (zusammenhängende Dateien (*contiguous files*) für kalkulierbare Zugriffszeiten)
- Write-Once Dateisysteme (CD/DVD, Logs, Backups, Versionierung).

Notizen

Allokation mittels verketteter Liste

Idee: Speicherblöcke einer Datei werden durch Verweise miteinander verkettet.

Jeder Block hat einen **Verweis auf Nachfolger-Block**

Verweis z.B. direkt am Beginn jedes Speicherblocks

Verzeichniseintrag verweist auf ersten Block der Datei

Vorteile:

- Keine Externe Fragmentierung

Nachteile:

- Wahlfreier Zugriff ist seeehr langsam.
- Es steht nicht der gesamte Datenblock für Daten zur Verfügung.

Notizen

Allokation mittels verketteter Liste und Index

Hochschule RheinMain

Speicherung der Verkettungsinformation in einer separaten Tabelle (Index oder **F**ile **A**llocation **T**able = FAT).

Vorteile:

- Gesamter Datenblock steht für Daten zur Verfügung.
- Akzeptable Performance bei direktem Zugriff, da der Index im Arbeitsspeicher gehalten werden kann.

Nachteile:

- Die gesamte Tabelle muss im Arbeitsspeicher gehalten werden. Kann bei großer Platte sehr speicherplatzaufwändig sein.

Beispiel: MS-DOS FAT File System

Notizen

Allokation mittels Index Nodes (1)

Hochschule RheinMain

Definition:I-Node

Ein **I-Node** (UNIX: *inode*) oder Index Node ist ein Dateikontrollblock

enthält neben Attributen der Datei eine Tabelle mit Adressen von zugeordneten Plattenblöcken.

Ursprung Beispiel: BSD UNIX Fast File System (ufs)

Notizen

Allokation mittels Index Nodes (2)

Hochschule RheinMain

Logische Blocknummern einer Datei ...

...werden fortlaufend vergeben,
beginnend bei den direkt adressierten Blöcken.

Einige wenige (~12) Blockadressen sind im Inode selbst gespeichet

→ Nach Öffnen einer Datei und damit verbundenem Einlagern des Inodes in den Hauptspeicher stehen diese Adressen sofort zur Verfügung.
→ Schneller Zugriff bei kleinen Dateien.

Für größere bis sehr große Dateien werden nach und nach einfach, zweifach und dreifach indirekte Blöcke zur Speicherung verwendet.

→ Zugriffsgeschwindigkeit sinkt bei größeren Dateien.

Beispiel: Linux ext2, ext3

Blockgröße: 4KB
12 direkt adressierte Blöcke → 48KB
Es sind 1-fach, 2-fach und 3-fach indirekte Blöcke vorgesehen
Indirekte Blöcke (4KB) speichern bis zu 1024 ($=2^{10}$) weitere Verweise
⇒ Maximale Dateigröße:
$(12 + 2^{10} + 2^{30} + 2^{30}) \cdot 4KB = 48KB + 4MB + 4GB + 4TB \approx 4TB$

Notizen

Allokation mittels Index Nodes (3)

Hochschule RheinMain

Nutzung von Indirekt-Blöcken

Notizen

Implementierung von Verzeichnissen

Hochschule RheinMain

Hauptaufgabe des Verzeichnissystems:
Abbildung der Zeichenketten-Namen von Dateien in Informationen zur Lokalisierung der zugeordneten Plattenblöcke.

Bei Pfadnamen werden die Teilnamen zwischen Separatoren schrittweise über eine Folge von Verzeichnissen umgewandelt.

Verzeichniseintrag liefert bei gegebenem Namen (Teilnamen) die Information zum Auffinden der Plattenblöcke:

- bei kontinuierlicher Allokation: die Plattenadresse der gesamten Datei oder des Unterverzeichnisses.
- bei Allokation mit verketteter Liste mit und ohne Index: die Plattenadresse des ersten Blocks der Datei oder des Unterverzeichnisses.
- bei Allokation mit Index Nodes: die Nummer des Inodes der Datei oder des Unterverzeichnisses.

Notizen

Beispiel: MS-DOS

Hochschule RheinMain

(s.o.) Hierarchisches Verzeichnissystem, Allokation von Plattenblöcken mittels verketteter Liste und Index.
Verzeichniseintrag:

Notizen

Beispiel: UNIX (1)

Hochschule RheinMain

(s.o.) Hierarchisches Verzeichnissystem, Allokation von Plattenblöcken mittels Index Nodes (inodes).

Verzeichniseintrag Linux (ext2:)

Verzeichniseintrag klassisches UNIX System V (s5):

Notizen

Beispiel: UNIX (2)

Hochschule RheinMain

Prinzip der Umsetzung eines Pfadnamens

Notizen

Dateisystemstruktur

Hochschule RheinMain

Die Struktur eines Dateisystems wird beim Erzeugen auf die Blockmenge eines logischen Laufwerks (Partition) aufgeprägt.

Dienstprogramme zum Erzeugen:

- MS-DIS: `format`[5]
- UNIX: `mkfs`, (`newfs`)

Beispiel: MS-DOS

[5] nicht zu verwechseln mit dem Formatieren eines Mediums, in MS-DOS low-level Formatierung genannt

Notizen

Dateisystemstruktur (2)

Hochschule RheinMain

Beispiel: Klassisches UNIX System V (s5)

Notizen

Verwaltung freier Blöcke

Hochschule RheinMain

Angewendete Methoden:

Verkettete Liste.

- Vorteil: Größere freie Bereiche einfacher erkennbar
- Beispiele: MS-DOS FAT, UNIX System V

Bitmap

- Vorteil: Größere freie Bereiche einfacher erkennbar.
- Beispiel: Linux ext2

Notizen

Dateisystemstruktur (3)

Hochschule RheinMain

Beispiel: Linux ext2: Besonderheiten

Einführung sogenannter **Bockgruppen**, d.h. Mengen von aufeinander folgenden Blöcken innerhalb eines Dateisystems mit jeweils eigenen Verwaltungsstrukturen.

I-Node und zugehörige Datenblöcke sollen möglichst dicht beisammen bleiben (Performance).

Redundante Kopien s1, .. des Superblocks in jeder Blockgruppe an verschiedenen Stellen (Kopfpositionen) zur Verbesserung der Verfügbarkeit der Layoutinformation auch bei Plattenfehlern.

Notizen

Dateisystemstruktur (4)

Hochschule RheinMain

Weitere Ansätze (hier nicht im Detail besprochen)

Log-basierte Dateisysteme:

- Für Halbleiter-Speichermedien konzipiert (keine seek-Zeiten)
- Schreibaufträge puffern, in regelmäßigen Zeitabständen als ganzes Segment schreiben.
- Position der inodes über in-Memory Map verwalten.
- Platte wird als „Ringpuffer“ betrieben.
- „Cleaner“ Thread gibt regelmäßig unbenutzte Blöcke frei.

Journaling-Dateisysteme:

- Erst: Geplante Operationen (idempotent) in Log schreiben
- Dann: Operationen ausführen
- Bei Absturz: Geplante, aber nicht mehr zur Ausführung gekommene Operationen nachholen.

Notizen

Quotas (Plattenplatz-Kontingentierung)

Hochschule RheinMain

Ziel:
Vermeidung der Monopolisierung von Plattenplatz durch einzelne Benutzer in Mehrbenutzersystemen.

Systemadministrator kann jedem Benutzer **Schranken** (Quotas) zuordnen für

- maximale Anzahl von eigenen **Dateien** und
- Anzahl der von diesen benutzten **Plattenblöcke**

Betriebssystem stellt sicher, dass diese Schranken nicht überschritten werden.

Beispiel: UNIX Quotas

Je Benutzer und Dateisystem werden verwaltet:

- Weiche Schranke (Soft Limit) für die Anzahl der benutzten Blöcke (kurzfristige Überschreitung möglich).
- Harte Schranke (Hard Limit) für die Anzahl der benutzten Blöcke (kann nicht überschritten werden).
- Anzahl der aktuell insgesamt zugeordneten Blöcke.
- Restanzahl von Warnungen. Diese werden bei Überschreitung des Soft Limits beim Login beschränkt oft wiederholt, danach ist kein Login mehr möglich.
- Gleiche Information für die Anzahl der benutzten Dateien (Inodes).

Notizen

Wahl der Blockgröße

Hochschule RheinMain

Fast alle Dateisysteme bilden Dateien aus Blöcken fester Länge
(Block umfasst zusammenhängende Folge von Sektoren.)
Problem: Welches ist die optimale Blockgröße?

Kandidaten aufgrund der Plattenorganisation sind:

- Sektor (512 B - 4KB)
- Spur (z.B. 256 KB)

Untersuchungen an Dateisystemen in UNIX-Systemen zeigen:

- Die meisten Dateien sind klein (< 10 KB) aber mit wachsender Tendenz.
- In Hochschul-Umgebung im Mittel 1 KB [Tanenbaum]

Eine große Allokationseinheit (z.B. Spur) verschwendet daher zuviel Platz.

Beispielrechnung: Verschwendeter Platz

(Daten basieren auf realen Dateien, Quelle [Leffler et al].)

Gesamt [MB]	Overhead [%]	Organisation
775.2	0.0	nur Daten, byte-variabel lange Segmente
807.8	4.2	nur Daten, Blockgröße 512 B, int. Fragmentierung
828.7	6.9	Daten und Inodes, UNIX System V, Blockgröße 512 B
866.5	11.8	Daten und Inodes, UNIX System V, Blockgröße 1 KB
948.5	22.4	Daten und Inodes, UNIX System V, Blockgröße 2 KB
1128.3	45.6	Daten und Inodes, UNIX System V, Blockgröße 4 KB

Notizen

Wahl der Blockgröße (2)

Hochschule RheinMain

Eine kleine Allokationseinheit (z.B. Sektor) führt zu schlechter zeitlicher Performance (viele Blöcke = viele Kopfbewegungen).

Mechanische Festplatte

https://medium.com/@duhroach/the-impact-of-blocksize-on-persistent-disk-performance-7e50a85b2647

zum Vergleich: SSD
N.B.: andere Skalierung

https://medium.com/@duhroach/the-impact-of-blocksize-on-persistent-disk-performance-7e50a85b2647

Notizen

Wahl der Blockgröße (3)

Hochschule RheinMain

Kompromiss:

Wahl einer mittleren Blockgröße, z.B. 4 KB oder 8 KB.
Bei Sektorgröße 512 B entspricht ein Block von 4 KB Größe dann 8 aufeinanderfolgenden Sektoren.
Für Lesen oder Schreiben eines Blockes wird die entsprechende Folge von Sektoren als Einheit gelesen oder geschrieben.
Ab ca. 2010 für PC-Systeme und Notebooks vermehrt Festplatten mit 4 KB-Sektorgröße (*Advanced Format*)

Ca. 9% Kapazitätsgewinn (da weniger Gaps)
Kompatibilitätsprobleme (Lösung durch Emulation) können Performance-Nachteile haben

Notizen

Wahl der Blockgröße (3)

Hochschule RheinMain

Tricks ...

Beispiel: BSD UNIX Fast File System (heute ähnlich auch bei ext2)

Einführung **zweier** Blockgrößen, genannt **Block** und **Fragment** als Teil eines Blocks (typ. heute 8 KB / 1 KB).

Eine Datei besteht aus ganzen Blöcken (falls nötig) sowie ein oder mehreren Fragmenten am Ende der Datei.

→ Transfer großer Dateien wird effizient.

→ Speicherplatz für kleine Dateien wird gut genutzt.
Empirisch wurde ein ähnlicher Overhead für ein 4KB/1KB BSD File System beobachtet wie für das 1 KB System V File System.

Notizen

Repräsentierung im Hauptspeicher

Hochschule RheinMain

Bisher wurde die Repräsentierung von Dateien und Verzeichnissen auf dem Hintergrundspeicher betrachtet.

Geöffnete Dateien besitzen eine Repräsentierung im Arbeitsspeicher

N.B.: Die System-Tabelle enthält insbesondere die Datei-Position jeder geöffneten Datei. Da geöffnete Dateien bei fork() vererbt werden, ist die einmalige Verarbeitung des Dateiinhalts durch Eltern- und Kindprozess oder durch mehrere Kindprozesse möglich. Erneutes Öffnen derselben Datei resultiert dagegen in einem neuen Eintrag mit unabhängigem Positionszeiger.

Notizen

Virtuelles Dateisystem

Hochschule RheinMain

Innerhalb des BS-Kerns wurde neue Schnittstelle des Dateisystems eingezogen, das sogenannte vnode-Interface (virtual inode) des VFS (Virtual File System).
Das vnode-Interface umfasst generische Operationen zum Umgang mit Dateien und Verzeichnissen bzw. Dateisystemen als ganzes.
Die virtuelle Schnittstelle wird für jeden Dateisystemtyp implementiert.
Das Interface ist auch auf Pseudo-Dateisysteme anwendbar, wie etwa das `/proc`-Prozessdateisystem (jedem aktiven Adressraum entspricht eine Datei) oder RAM-Disk.

Notizen

Zuverlässigkeit des Dateisystems

Hochschule RheinMain

Hauptgesichtspunkte:

Behandlung fehlerhafter Blöcke.

Dateisystemkonsistenz.

Erzeugen und Verwalten von Backups.

Notizen

Behandlung fehlerhafter Blöcke

Hochschule RheinMain

Sowohl Festplatten als auch Flash-Speicher haben i.d.R. von Anfang an **fehlerhafte Blöcke** (z.B. aufgrund ungleichmäßiger Magnetisierung der Oberflächen, Toleranzen in der Chip-Fertigung, etc).
Bei Festplatten werden fehlerhafte Sektoren beim Formatieren des Mediums (Aufbau der Sektoren) festgestellt. (Im PC-Umfeld wird das Formatieren auch low-level-Formatierung genannt).
Bei Flash-Speichern kommen aufgrund des Verschleißes im lfd. Betrieb weitere defekte Blöcke hinzu.
Verzeichnis der fehlerhaften Sektoren wird **Media Defect List** genannt.
Hardware-Lösung (heute üblich):

- Media Defect List wird vom Gerätecontroller selbst geführt.
- Jedem defekten Block wird ein Ersatzblock (i.d.R. aus einem dafür reservierten Bereich) zugeordnet, der statt des defekten Blocks benutzt wird.
- Bei Festplatten Problem: evtl. unvorhersehbare Kopfbewegungen.
- Bei Flash-Speichern kann der reservierte Bereich irgendwann erschöpft sein.

Software-Lösung (heute eher unüblich):

- Es wird eine Datei konstruiert, die nie gelesen oder geschrieben wird und der alle defekten Blöcke zugeordnet werden.

Notizen

Konsistenz des Dateisystems

Hochschule RheinMain

Definition: Dateisystem-Konsistenz

Konsistenz eines Dateisystems meint Korrektheit der inneren Struktur des Dateisystems.

d.h. aller mit der Aufprägung der Dateisystemstruktur auf die Blockmenge verbundenen Informationen (Meta-Information des Dateisystems, UNIX: z.B. Superblock, Freiliste oder Bitmap).

Beispiel einer Konsistenzregel:

- Jeder Block ist entweder Bestandteil genau einer Datei oder eines Verzeichnisses, oder er ist genau einmal als freier Block bekannt.

Verletzung der Konsistenz:

- I.d.R. durch Systemzusammenbruch (z.B. aufgrund eines Stromausfalls) vor Abspeicherung aller modifizierten Blöcke eines Dateisystems.

Überprüfung der Konsistenz:

- Betriebssysteme besitzen Hilfsprogramme zur Überprüfung und evtl. Wiederherstellung der Konsistenz bei eventuell auftretendem Datenverlust.

Notizen

Beispiel: UNIX

Hochschule RheinMain

UNIX war traditionell schwach in Bezug auf Sicherstellung der Konsistenz von Dateisystemen:

Dateisystem (z.B. ufs) wird **nicht in atomaren Schritten** von einem konsistenten Zustand in einen neuen konsistenten Zustand überführt. Eine solche Veränderung verlangt i.d.R. mehrere Schreibzugriffe.

Modifizierte Datenblöcke bleiben im Pufferspeicher (Block Buffer Cache) und werden durch einen Dämonprozess spätestens nach 30 sec zurückgeschrieben.

Modifizierte Blöcke mit Meta-Informationen werden zur Verringerung der Gefahr der Inkonsistenz sofort zurückgeschrieben.

System Call `sync` existiert zur Einleitung eines sofortigen Zurückschreibens aller veränderten Blöcke (*forced write*).

Notizen

Beispiel: UNIX (2)

Hochschule RheinMain

Durch Einführung von Journaling-Dateisystemen[6] hat sich die Situation deutlich gebessert.

- Meta-Informationen des Dateisystems werden vorab per Write-Ahead-Logging (analog Datenbanken) gespeichert.
- Konsistenz ist gewährleistet, z.B. bei Systemzusammenbruch.
- Dennoch können plötzliche Stromausfälle während laufender physischer Schreibvorgänge Schäden verursachen.
- Gilt für Festplatten wie für Flash-Speicher.
- Deshalb: Medien vor dem Entfernen immer erst „unmounten“.

Zur Verbesserung der Verfügbarkeit von Daten im Falle von Plattenfehlern werden z.B. eingesetzt:

- Spiegelplattenbetrieb (RAID1 - Disk mirroring, vgl. 10.3.2)
- Paritätsinformation speichern (RAID5, vgl. 10.3.2)

[6] `ext3`, `ext4`, ReiserFS, Btrfs, XFS, ..., vgl 10.3.5

Notizen

Überprüfung und Reparatur

Hochschule RheinMain

Dienstprogramme zur Überprüfung / Reparatur der Konsistenz eines **nicht in Benutzung** befindlichen Dateisystems **bei möglichem Datenverlust**: `fsck`, z.T. auch `ncheck`

`fsck` (*file system check*) führt Konsistenzüberprüfungen durch:

Blocküberprüfung:

- zwei Tabellen mit jeweils einem Zähler je Block
- anfangs alle Zähler mit 0 initialisiert

0					5					10					15	
1	1	0	1	1	1	0	0	0	0	0	1	1	0	1	0	belegte Blöcke
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1	freie Blöcke

- erste Tabelle: wie oft tritt jeder **Block in einer Datei** auf?
  - alle inodes lesen.
  - für jeden verwendeten Block Zähler in erster Tabelle inkrementieren.
- zweite Tabelle: **freie Blöcke**
  - Für Blöcke in der Liste / Bitmap der freien Blöcke Zähler in zweiter Tabelle inkrementieren.

Konsistenz: Für jeden Block muss der Zählerstand aus Tab 1 und Tab 2 zusammen „1“ sein.

Notizen

Überprüfung und Reparatur (2)

Hochschule RheinMain

Fehlender Block: Block 4 ist weder belegt noch frei?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	0	0	0	0	0	1	1	0	1	0	belegte Blöcke
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1	freie Blöcke

→ Maßnahme: Block zu freien Blöcken hinzunehmen

Doppelter Block in Freiliste (Block 8)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	1	1	0	0	0	0	0	1	1	0	1	0	belegte Blöcke
0	0	1	0	0	0	1	1	2	1	1	0	0	1	0	1	freie Blöcke

→ Maßnahme: Freiliste neu aufbauen

Doppelter belegter Block (Block 11)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	1	1	0	0	0	0	0	2	1	0	1	0	belegte Blöcke
0	0	1	0	0	0	1	1	1	1	1	0	0	1	0	1	freie Blöcke

→ Maßnahme: Block kopieren, Kopie-Block in eine der beiden betroffenen Dateien statt Block 11 einbauen

Notizen

Überprüfung und Reparatur (3)

Hochschule RheinMain

Darüber hinaus **Überprüfung der Verzeichniseinträge**:

- Vergleiche Anzahl aller Verzeichniseinträge mit Verweis auf einen inode mit dem darin gespeicherten Referenzzähler.

→ Maßnahme: ggf. inode-Referenzzähler der durch Zählung festgestellten Zahl von Referenzen anpassen.
(N.B.: Es kann „beliebig viele“ (> 0) Referenzen auf einen inode geben (aufgrund harter Links!))

Problem: Für große Platten kann ein `fsck`-Lauf sehr lange (Stunden!) dauern. In dieser Zeit ist das System möglicherweise nicht verfügbar (Kosten!)

Notizen

Performance des Dateisystems

Hochschule RheinMain

Maßnahmen zur Performance-Steigerung:

Blockgruppen:

- Ziel: Vermeiden von weiten Kopfbewegungen.
- Beispiel: Linux ext2 File System (vgl. 10.3.5).
- Kein Effekt (auch kein negativer) bei Flash-Speicher

Block Buffer Cache.

- Ziel: Reduzierung der Anzahl der Plattenzugriffe.
- Ein Teil des Arbeitsspeichers, **(Block) Buffer Cache** genannt, wird als Cache für Dateisystemblöcke organisiert.
- Typische Hitrate: 85%
- Modifizierter LRU-Algorithmus zur Auswahl zu verdrängender Blöcke unter Berücksichtigung der Forderungen zur Verringerung der Gefahr von Inkonsistenz
- Blöcke verbleiben im Cache, auch wenn die entsprechenden Dateien geschlossen sind.

Notizen

Performance des Dateisystems (2)

Hochschule RheinMain

Dateinamens-Cache:

- Ziel: Verringerung der Anzahl der Schritte bei der Abbildung von Datei-Pfadnamen auf Blockadressen.
- Beispiel: BSD UNIX namei-Cache:
  - `namei()`-Routine
    zur Umsetzung von Pfadnamen auf inode-Nummern.
    benötigte vor Einführung von Caching ca. 25% der Zeit eines Prozesses im BS-Kern, auf <10% gesenkt.
  - Cache der *n* letzten übersetzten Teilnamen:
    Typische Hitrate: 70-80%.
    Höchste Bedeutung für Gesamtperformance.
  - Cache des letzten benutzten Directory-Offsets:
    Wenn ein Name im selben Verzeichnis gesucht wird, beginnt die Suche am gespeicherten offset und nicht am Anfang des Verzeichnisses (sequentielles Lesen eines Verzeichnisses ist häufig, z.B. durch das ls-Kommando).
    Typische Hitrate: 5-15 %.
  - Gesamthitrate von ca. 85% wird erreicht.

Notizen

Zusammenfassung

Hochschule RheinMain

Was haben wir in Kap. 10 gemacht?

Konzepte von Dateisystemen.
Verwalten von Mengen von Dateien und Verzeichnissen.

- Zunächst eine äußere Benutzersicht
  - Benennung von Dateien
  - Dateistrukturen und -Typen
  - Zugriffsarten
  - Dateiattribute
  - Hierarchische Strukturierung.
- Dateisysteme aus der Sicht der Implementierung
  - Technische Gegebenheiten
  - Plattenplatz-Verwaltung
  - Implementierung von Dateien
  - Implementierung von Verzeichnissen
  - Dateisystemstruktur
  - Plattenplatz-Kontingentierung
  - Wahl der Blockgröße
  - Repräsentierung im Hauptspeicher
  - Zuverlässigkeit des Dateisystems
  - Performance des Dateisystems

Notizen

Notizen