Betriebssysteme I

Zusammenfassung v1.1

Kälin Thomas, Abteilung I
WS 05/06

1. Grundlagen

1.1. Zweck

- Erweiterte Maschine
  - verbirgt viele kleine, applikationsunabhängige Teilfunktionen
  - Kann einfacher benutzt & programmiert werden als blanke Rechner
- Betriebsmittelverwalter
  - verwaltet zeitliche Zuteilung der Ressourcen
  - verwaltet räumliche Zuteilung der Ressourcen

1.2. Funktionen

- Hardwareunabhängige Programmierschnittstelle, Geräteunabhängige Ein/Ausgabe-Funktionen, Ressourcenverwaltung (Aktive Ressource), Speicherverwaltung (Passive Ressource), Massenspeicherverwaltung, Multitasking, IPK, Sicherheitsmechanismen
- Bindeglied zwischen Hardware und Applikationen

1.3. Definition

- Ein Betriebssystem ist eine Menge von Programmen, die die Ausführung von Benutzerprogrammen auf einem Rechner und den Gebrauch der vorhandenen Betriebsmittel steuern

2. Programmausführung und Hardware

2.1. Grundmodell eines Rechners

- Von-Neumann-Rechner: Seite 23 im Buch

2.2. Befehlsverarbeitung der CPU

- FETCH / EXECUTE (Seite 24)

2.3. Prozessoraufbau

2.3.1. Leitwerk (Steuerregister)

- PC: Momentaner Ausführungspunkt
- SP: Zeigt aktuelles oberes Ende des Stapels an (TOS)
- PSW: Zeigt Prozessorstatus an, teilweise aber auch Steuerfunktionen
  - **Kernelmode:** privilegierte Betriebsart, nur für OS, alle Instruktionen erlaubt
  - **Usermode:** nicht-privilegierte Betriebsart, für Anwendungen Einschränkung der Instruktionen & Register
  - Umschaltung in Kernmodus: SW oder HW-Interrrupt, PR-Flag auf 0 setzen
  - Umschaltung in Benutzermodus: Maschinenbefehl, PSW-Modifikation

2.3.2. Rechenwerk (ALU) und Registerblock (Allgemeine Register)

- Prozessorabhängig
- Registerbreite in xx Bit -> bestimmt xx-Bit-Architektur
- Adressierbarkeit
- Funktion: Zwischenspeicherung von Operanden und Resultaten

2.4. Adressraum (30)

- Adresse = Hausnummer, i. d. R. 1 Byte pro Speicherstelle
- Adressbus = 32Bit (Intel)
- Adressraum = Menge aller möglichen Adressen = {0,1,2,…,2^*32*-1}
  - Adressraum Intel Pentium-Prozessor: 4 GB
  - E/A-Adressraum: 64kB
- Adressraumgrösse = Anzahl aller möglichen Adresse = 2^*32*
- Memory mapped I/O: E/A-Bausteine liegen im Hauptspeicherraum und erhalten eine Hauptspeicheradresse von der CPU

2.4.1. Klassische Von-Neumann-Rechner (31)

- Es ist ein einziger Hauptspeicheradressraum für Daten, Programme und Ein-/Ausgabe vorhanden.

2.4.2. Erweiterte Von-Neumann-Rechner (31)

- Intel Pentium Prozessor
- Es existiert ein Hauptspeicheradressraum für Daten und Programme (4GB bei Intel). Zusätzlich steht aber noch ein kleiner Ein-/Ausgabeadressraum (64kB bei Intel) zur Verfügung.
- Nachteil: Grösserer HW-Aufwand
- Vorteil: Höherer Instruktionsdurchsatz

2.4.3. Unterschiede

- Die Daten und Instruktionen werden beim Harvard-Rechner über getrennte Datenpfade aus getrennten Daten- bzw. Instruktionsspeichern transportiert. Beim Von-Neumann-Rechner sind die Daten und Instruktionen im selben Speicher abgelegt und werden über einen einzigen Datenpfad transportiert.

2.5. Bytereihenfolge

- Wichtig beim Dateiaustausch zwischen PC und MAC

```
0x080490BA:  0x78  0x56  0x34  0x12
```

2.5.1. Big Endian – Power PC / Motorola (33)

```
0x78563412
```

2.5.2. Little Endian – Intel Pentium(33)

```
0x12345678
```

2.6.Ausrichtungsregeln (35)

- Legen fest, auf welchen Adressen Variablen und Instruktionen liegen
- Zweck: Erreichung optimaler Ausführungsgeschwindigkeit
- Datenbusbreite 32 Bit: optimale Ausrichtung für 4-Byte-Werte (immer 4-Byte in einem Zyklus)

2.6.1. Misalignment (36)

- Zwei Zugriffszyklen anstatt einem

2.6.2. Einfluss auf Adressraumplatzierung (36)

- Es entstehen „Lücken" (int-Variable mit 2 Byte, bei alignment(4) entstehen 2 Byte-Lücke)
- Wichtig bei Debugger-Analyse!
- Beispiel: sizeof(struct); -> 11 Byte erwartet, 16 Byte bekommen. Wegen diesen Lücken!

2.7.Adressraumbelegung (37)

- Code / Konstanten, init. Daten, nicht init. Daten, Heap, Stack (Siehe Schema im Buch!)

2.8.Grundlagen der Programmausführung (40)

2.8.1. Operandenspeicherung / Assembler / xx-Adress-Maschinen (44)

- Null-Adress-Maschine, Ein-Adress-Maschine, Zwei-Adress..

2.8.2. Programmzähler: PC (44)

- Beschreibt momentanen Ausführungspunkt, befindet sich in CPU
- Wir mit jedem Befehlszyklus um 1 erhöht, zeigt jeweils auf die nächste auszuführende Instruktion im Speicher, enthält also eine Speicheradresse.
- Bei grösseren Anweisungen (2 bis 10Byte) wird jeweils um diesen Wert erhöht.
- Heisst bei Intel-Pentium „eip"

2.8.3. Stapelspeicher, Stapelzeiger, Stackframe (47-60)

- Stackframes werden immer oben hingelegt.
- Stackpointer (esp) zeigt immer auf obersten Eintrag, TOS)
- Ein Aktivierungsrahmen (Stackframe) enthält folgende Elemente:
  - Aufrufparameter (von links nach rechts beim GNU-Compiler)
  - Rücksprungadresse in Oberprogramm
  - Alter Wert des **Framepointers (ebp)**, Zeigt auf Bereich lokaler Variablen -> Einfacher Zugriff
  - Lokale Variablen
  - Nach Bedarf weitere zwischengespeicherte Werte

```
Beispiel eines Stackframes (Buch S57)
0xbffff578   0x61  0x84  0x04  0x08  //Platz für lokale Variablen
0xbffff57C   0x88  0xf5  0xff  0xbf  //Alter Wert des ebp (0xbffff588)
0xbffff580   0xf1  0x83  0x04  0x08  //Rücksprungadresse (0x080483f1)
0xbffff584   0x44  0x33  0x22  0x11  //Parameter, int (0x11223344)
```

2.8.4. Parameterübergabe via globale Variable (Stack-Übung: E5)

- Vorteile
  - Kein Umkopieren von Werten, weniger Code, Variablen überleben Unterprogramm, kein Platz für Parameter auf Stapel
- Nachteile
  - Unerwünschte Nebeneffekte, Code kann verwirrend sein

3. Systemprogrammierung (61)

- Systemprogrammierung = betriebssystemnahe Programmierung
- Direkte Nutzung der generischen Systemprogrammierschnittstellen (native API)

3.1. Warum Systemprogrammierung?

- Pro: Zugriff auf systemspezifische Optionen, Zugriff auf spezielle Systemdienste, eine Softwareschicht weniger (Effizienz)
- Contra: Nutzung von Sprachbibliotheken (Java) meist einfacher

3.2. Dienstanforderung & Erbringung (65)

3.3. Dienstparameter (66)

- Möglichkeiten: Parameter, Attributmengen, Structs

3.4. Rückgabewerte (69)

- Möglichkeiten: Rückgabewert der Funktion, Zeiger

3.5. Opake Datentypen (76)

- interne Struktur soll unsichtbar bleiben
- Systemdatentypen = opake Datentypen
- Ziel: portable, änderungsfeste Quellprogramme
- Opak ist das Gegenteil von transparent, d.h.: Der Datentyp soll vor dem Anwender versteckt werden, da er nur für das Betriebssystem eine Bedeutung hat. Es soll auch die Möglichkeit bestehen, diesen Datentyp betriebssystemseitig zu ändern, ohne dass der Quellcode der Applikationsprogramme angepasst werden muss.

3.6. Umgebungsvariablen (70)

- Aufbau: Liste von Paaren, bestehen aus Textinformationen
- Nutzen: Konfiguration von Programmen

3.7. Dateideskriptoren & Handles (73)

- Windows -> Handles, Unix -> Deskriptor
- Benutzerfreundlich und systemfreundlich

4. Prozesse & Threads (88)

4.1. Parallelverarbeitung

- Definition: mehrere Aktivitäten können nebeneinander, d.h. gleichzeitig stattfinden
- Parallele Abläufe sind in Hardware (90) und Software (91) möglich

4.1.1. Begriffe (92)

- Prozess = Programm in der Ausführung
  - belegt Hauptspeicherplatz, verbraucht Rechenzeit, …
  - „kochen"
  - Was ist ein Prozess? – Aus Sicht des Betriebssystems: Datenbereich (inkl. Stack und Heap), Programmcodebereich, PCB (Process Control Block)
  - Jeder Prozess hat seinen eigenen abgegrenzten Speicherbereich
- Programm = Verfahrensvorschrift für elektronische Datenverarbeitung
  - belegt Plattenplatz und enthält Programm- / Ausführungsinstruktionen
  - „Kochrezept"
- Echtzeitbetrieb
  - Ist ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, so dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind.
- Thread
  - Eigene Ablaufstränge in gemeinsamer Ablaufumgebung
  - Zu einem Prozess gehörige Threads teilen sich den Adressraum untereinander

4.2. Prozessmodell (94)

- Jeder Prozessor hat virtuell den ganzen Rechner für sich alleine zur Verfügung
  - Kann Hauptspeicher frei belegen
  - Kann Register der CPU frei benutzen
  - Sieht nur seine eigenen Daten

4.2.1. Mehrprogrammbetrieb

- Aufteilung der Rechenzeit auf mehrere ablaufwillige Prozesse
  - Illusion der echten parallelen Programmausführung (pseudoparallel)

4.2.2. Prozessumschaltung (96)

- PCB = Process Control Block
  - Enthält alle Daten, die das OS über einen Prozess zur Verwaltung führen muss: PID, Prozesszustand, Program Counter, Stack Pointer, Prozesskontext, Filedeskriptoren-Tabelle (u.v.m.)
- 3 Schritte
  - Kontextsicherung aktueller Prozess
  - Auswahl nächster Prozess
  - Kontextwiederherstellung neuer Prozess

4.2.3. Prozesserzeugung und Terminierung (98)

- Einfache Systeme (embedded systems): Alle Prozesse existieren von Anfang an und laufen ewig
- Allgemeinere Systeme: Prozesse werden gestartet und wieder terminiert
- **Prozessverkettung**: Laufender Prozess startet einen neuen und terminiert sich selbst
  - chain
- **Prozessvergabelung**: Laufender Prozess startet einen neuen und läuft selber weiter
  - fork / join .. exit
- **Prozesserzeugung**: Der laufende Prozess startet einen neuen, unabhängigen Prozess (Thread)
  - create / wait .. exit
- Weitergabe der Daten
  - Vererbung, Weitergabe über Datenpuffer, Initialparameter, temporäre Daten

4.2.4. Prozessvergabelung: Forking (102)

```
int main() {
 int k, status;
 pid_t pid;
```

```
 k = fork();
 if (k == 0) {
       //Anweisungen für den Kindprozess
       exit(0);
 } else {
       //Anweisungen für den Elternprozess
       pid = wait(&status);
       //Aufrufender Prozess wird blockiert, bis IRGENDEIN Kind terminiert
       //Mit Marko WIFEXITEX(status) Prüfung auf normale Terminierung
       //Mit Marko WEXITSTATUS(status) -> Wert von exit(x) abrufen
       exit(0);
 }
}
```

- Zombieprozess: Entsteht, wenn ein Kindprozess terminiert, bevor der Elternprozess zu warten beginnt. Dieser Prozess führt keinen Code mehr aus und existiert in diesem Sinne nicht mehr. (104)
- Waise (orphan): Kindprozess, dessen Elternprozess nicht mehr existiert. Erhält als neuen Elternprozess den init-Systemprozess (PID=1)
- Der Kindprozess erbt: Code und Kopie aller Daten, Dateideskriptoren des Elternprozess, Umgebungsvariablen, Arbeitsverzeichnis (114)

4.2.5. Verkettung: Chaining (106)

- Ein erfolgreicher exec()-Aufruf hat zur Folge, dass der neue Prozess startet und gleichzeitig der alte Prozess terminiert.
- Sechs varianten von exec(). Diese unterschieden sich jedoch nur in den verfügbaren Funktionsparametern, nicht in ihrer Grundfunktion.
- Der neue Prozess erbt Adressraum, PID, Deskriptoren, Eltern-Kind-Beziehung, Umgebungsvariablen, Arbeitsverzeichnis (114)

4.2.6. Grundfunktion der Unix-Shell (109)

- Siehe Buch

4.2.7. Prozesse & Jobs unter Windows (111)

- Windows unterstützt nur eine einzige Variante für den Prozessstart: Prozesserzeugung
- Der neue Prozess ist unabhängig vom alten

```
CreateProcess()
```

- Nur Handles, aber mit Bedingungen (114)

4.3. Threads (119)

- Threads beruhen auf der Idee der gemeinsamen Ressourcennutzung, d.h.: gemeinsamer Adressraum, gemeinsame Systemressourcen (geöffnete Dateien, …), gleiche globale Variablen.
- **Aber: Pro Thread vorhanden:** PC, Register, Stapel, PSW (TCB = Thread-Control-Block)
- Parallel ablaufende Aktivitäten in einem Prozess
- Vorteil: Schnellere Umschaltung zwischen Threads als zwischen Prozessen, da die Memory-Management-Unit nicht neu programmiert werden muss (gemeinsamer Adressraum)
- Vorteil: Einfache Nutzung gemeinsamer Ressourcen

4.3.1. Implementierung des Multithreading (122)

- User-Level-Thread (UL-Thread): Dies ist ein Thread, wie er für den Programmierer sichtbar ist. Wird eine Software entworfen, so legt man die Anwendung von UL-Threads fest, die eine gewünschte Parallelität logisch realisieren
- Kernel-Level-Thread (KL-Thread): Dies ist ein Thread, der tatsächlich auf der CPU abläuft. Der KL-Thread ist dem OS bekannt und erhält von diesem aufgrund einer bestimmten Strategie Rechenzeit zugeteilt.

4.3.2. m:1-Zuordnung (123)

- Alle zu einem Applikationsprozess gehörenden UL-Threads werden einem einzigen KL-Thread zugeordnet. Das MT findet hier vollständig ausserhalb des Systemkerns auf der Benutzerebene statt.

4.3.3. 1:1 Zuordnung (125)

- Jedem UL-Thread ist genau ein KL-Thread zugeordnet. Das Multithreading findet hier vollständig auf der Systemebene statt.

4.3.4. m:n-Zuordnung (126)

- Einem KL-Thread sind mehrere UL-Threads zugeordnet und es gibt mehrer KL-Threads pro Prozess. Dies ist eine Hybridlösung.

4.3.5. Codebeispiel Windows (127)

```
HANDLE hThread;
DWORD threadId;
int a = 5;
DWORD result;

hThread = CreateThread(NULL,0,ThreadFunc,&a,0,&threadId);
GetExitCodeThread(hThread,&result);
```

4.3.6. Systemaufrufe unter Windows (129)

Hinweis: Windows-Threads sind stets Kernel Level Threads. Bei Unix hängt dies von der Implementierung der POSIX-Programmbibliothek ab.

- Thread-Erzeugung

```
CreateThread()
```

- Thread-Terminierung

```
ExitThread() -> durch Thread selber
TerminateThread() -> durch anderen Thread
```

- Im PrimaryThread auf Terminierung des zweiten Threads warten

```
WaitForSingleObject(Handle,Zeit in ms)
WaitForMultipleObjects //Auf mehrere Threads warten, Maximal aber 64
```

- Beendigungsstatus

```
GetExitCode()
```

- Schliessen eines Thread-Handles

```
CloseHandle()
```

  - Ressource Leak: Es wird vergessen, die Systemfunktion CloseHandle() aufzurufen. Dadurch ist es nur eine Frage der Zeit, bis keine Systemobjekte mehr angelegt werden können, da die Systemressourcen erschöpft sind. Unter Umständen kann auf diese Art und Weise eine ganze Maschine lahm gelegt werden, im Minimum jedoch der Serverprozess selbst. Beim Beenden einer Anwendung, gibt das OS automatisch alle bezogenen Ressourcen wieder frei, unabhängig davon, ob ein CloseHandle()-Aufruf gemacht wurde.

4.3.7. Codebeispiel Unix (134)

- Siehe Buch…

4.3.8. Systemaufrufe unter Unix (135)

- Thread-Erzeugung

```
pthread_create() //Gibt 0 zurück, wenn Thread erfolgreich gestartet
```

- Auf Thread-Ende warten
- Kann von irgendeinem Thread aufgerufen werden

```
pthread_join() //2.Parameter ist Rückgabewert des Threads
```

- Thread-ID herausfinden

```
phread_self()
```

- Schliessen eines Threadhandels

```
pthread_detach()
```

5. Synchronisation von Threads & Prozessen (173)

5.1. Problem der Ressourcenteilung (173)

- Oft Zugriff auf gleiche Ressourcen durch parallele Aktivitäten
- Dies kann zu Synchronisationsprobleme führen

5.1.1. Lost update Problem (174)

- Es kann keine Annahme über das zeitliche Verhalten von Thread-Abläufen gemacht werden.
- Problem: Zwei Prozesse greifen gleichzeitig auf einen gemeinsamen Bereich (kritischen Bereich) zu. Eine Transaktion von beiden geht unweigerlich verloren.
- Lösung: Die Transaktionen müssen unteilbar (atomar) ablaufen
  - Mutual Exclusion: Wechselseitiger Ausschluss

5.1.2. Inkonsistente Abfrage (176)

- Problem: Wird ein Thread in seinem Schreibe-Vorgang unterbrochen (Rescheduling), so werden die Werte nur unvollständig geschrieben. Ein zweiter Prozess, welche auf diese Variable zugreift, bekommt inkonsistente Daten

5.1.3. Absicherung durch Selbstverwaltung

- Beispiel: boolsche Variable -> Keine Lösung, sondern nur Verschiebung des Problems

5.2. Semaphore (178)

5.2.1. Typen (180)

- Binärer Semaphor, Mutex: Es sind genau 0 oder 1 Marke erlaubt
- Zähl-Semaphor: Es sind beliebig viele Marken erlaubt

5.2.2. Operationen (unteilbar)

- P-Operation: Marke nehmen, vor kritischem Bereich
- V-Operation: Marke zurücklegen, nach kritischem Bereich

5.3. Anwendung der Sempahore (182)

5.3.1. Absicherung kritischer Bereich (Mutual Exclusion)

5.3.2. Synchronisation von Abläufen (184)

- Alle Semaphoren müssen mit 0 initialisiert sein

```
Thread A          Thread B          Thread C
V(synch_A);       V(synch_B);       V(synch_C);
V(synch_A);       V(synch_B);       V(synch_C);
P(synch_B);       P(synch_A);       P(synch_A);
P(synch_C);       P(synch_C);       P(synch_B);
```

5.3.3. Produzent / Konsument (186)

- Realisierung mittels Zählsemaphor

5.3.4. Leser und Schreiber (189)

- Nicht modifizierende Benutzung verursacht keine Probleme, schreibende jedoch schon
  - Lesen beliebig erlaubt, schreiben jedoch nur dann, wenn keiner liest

5.4. Vorteile / Nachteile (197)

- Vorteile:
  - Einfache Implementierung
  - Im BS praktisch immer vorhanden
  - Einfachstes Synchronisationsmittel
  - Anerkannter Grundmechanismus in Betriebssystemtheorie
- Nachteile
  - Code zur Synchronisation ist auf einzelne Prozesse verteilt, erschwert Übersichtlichkeit
  - Viele Fehlermöglichkeiten im logischen Programmentwurf (Deadlocks, überzählige V-Operation)
  - Wenig strukturierter Programmablauf

5.5.Deadlocks (225)

- Gruppe von Prozessen: Jeder wartet auf ein Ereignis, das nur ein anderer der Gruppe auslösen kann. Folge: Dauerhafte Blockierung
- Ursache: Gegenseitige Abhängigkeit, überlappende Reservierungen von Betriebsmitteln
- Ein Deadlock ist die Folge eines logischen Programmfehlers! Eine Deadlock-Gefahr muss daher bei der Programmentwicklung erkannt werden, beim Testen der SW besteht nur eine kleine Wahrscheinlichkeit, dass ein DL auftritt und erkannt wird
- Für das Ermitteln der Deadlockgefahr muss immer die Gesamtheit aller Prozesse beachtet werden

5.5.1. Betriebsmittelfahrplan (227)

- Nur geeignet, wenn es um zwei beteiligte Prozesse geht
- In einem Zweiprozessorsystem wären die Fahrbahnen nicht mehr strikt waagrecht oder senkrecht, sondern schräg von links oben nach rechts unten
- Deadlock-Gefahrt herrscht bei nach oben / links offenen Rechtecken
- Alle möglichen Ablaufpunkte auf einen Blick

5.5.2. Überlappungen (228)

- Einfache Methode, immer wenn sich bei einem Prozess zwei Striche überlappen, muss auf Verklemmungsfreiheit geprüft werden

5.5.3. Begriffe (229)

- Deadlock (siehe 5.5)
- Blockierung: Temporäre Blockierung eines Prozesses, weil er auf ein Ereignis wartet, das noch nicht aufgetreten ist (im Fehlerfall: Kann zu dauerhafter Blockierung führen)
- Dreschen: Die am Dreschen beteiligten Prozesse arbeiten, aber ihre Arbeit ist unproduktiv. (Beispiel: Viel Rechenzeit verbraucht für das Ein- / Auslagern von Hauptspeicherinhalten)
- Verhungern, Starvation: Ein Prozess wird am Weiterarbeiten gehindert, da er eine benötigte Ressource nie erhält, obwohl sie im Zeitablauf immer wieder frei wird. Grund: Andere Prozesse erhalten stets den Vorzug bei der Ressourcenzuteilung
- Aktives Warten: Es wird dauernd Rechenzeit verbraucht, um zu prüfen, ob die Wartebedingung erfüllt ist.
- Passives Warten: Der Prozess wird vom OS solange schlafen gelegt, bis das Warteereignis eintrifft

5.5.4. Deadlock-Bedingungen (230)

Ein Deadlock kann nur dann auftreten, wenn ALLE der folgenden 4 Bedingungen erfüllt sind

- Mutual Exclusion: Jede Ressource ist entweder genau einem Prozess zugeteilt (reserviert, abgesichert) oder allgemein erhältlich (nicht abgesichert)
- Hold and Wait Condition: Prozesse, die bereits mindestens eine Ressource besitzen, verlangen nach weiteren Ressourcen
- No Preemption: Zugeteilte (reserviert) Ressourcen können vom OS nicht zurückgefordert werden. Jeder Prozess gibt die Ress. nur dann zurück, wenn er sie aufgrund des logischen Ablaufs nicht mehr benötigt
- Zyklische Wartebedingungen: Es muss eine zyklische Kette von zwei oder mehr Prozessen geben, die alle auf eine Ressource warten, die vom Nachfolger bereits reserviert ist.

5.5.5. Lösungsansätze (230, 231)

- Deadlock-Ignorierung: Null-Lösung
- Deadlock-Vorbeugung: Lösung während Programmentwicklung
- Deadlock-Vermeidung: Lösung während Betrieb der SW
- Deadlock-Erkennung mit Auflösung: Lösung während Betrieb, aber mit dem Holzhammer
  - Prozessterminierungen

5.5.6. Nummerierung der Betriebsmittel (232)

- Zyklische Wartebedingung brechen
- Jedem Semaphor eine Zahl zuweisen. Wenn eine Nummerierung gefunden wird, bei der alle Prozesse die Ressourcen nur nach aufsteigenden Nummern reservieren, herrscht keine Deadlock gefahr.

5.5.7. Betriebsmittelgraph (233)

- Nur eine ganz bestimmte Situation beurteilen

6. Unix-Signale (214)

6.1.Zweck

- Unterbrechung eines Prozesses zur Behandlung eines speziellen Ereignisses

6.1.1. Verwendung

- Durch Unix Betriebssystem selber
  - Spezialfunktion (Wecker)
  - Zur Meldung von Fehlern an Benutzerprozesse (ungültiger Maschinenbefehl)
  - interaktiven Programmabruch (CTRL+C)
- Durch Benutzerapplikation
  - Zeitlimit mit Alarmsignalen setzen
  - Signalisieren anderer Prozesse

6.1.2. Auftreten

- synchron: Division durch 0
- asynchron: CTRL+C ab Tastatur

6.2.Signalgebrauch auf Kommandozeile (216)

- Mit Hilfe des Befehls kill / kill() können Signale an Prozesse gesendet werden

```
kill -<sigtype> <pid>
kill –SIGKILL <pid> //Terminierung des Prozesses

kill(pid,signr); //Systemaufruf
```

6.3.Programmierung der Signale (217)

6.3.1. Klassische signal-Funktion (217)

- Probleme
  - Aktueller Signalhandler kann nicht abgefragt werden
  - Nach Auftreten eines Signals wird automatisch wieder die Standardreaktion aktiviert. Kann zu Programmabbruch führen (zwei Mal hintereinander sehr schnelles Signal!)
  - Eintreffendes Signal wird stets sofort behandelt

```
void forward(int sigtype) {
 printf(„Bin geweckt worden!\n");
 signal(SIGALRM,forward); //Reaktion wieder setzen
}

int main(int argc, char *argv[]) {
 int i;
 sigignore(SIGKILL); //Ignoriere SIGKILL-Signale
 signal(SIGALRM,forward);
 while (1) {
       printf(„Schlafen…\n");
       alarm(3); //Sendet nach 3 Sekunden SIGALRM
       pause(); //Blockiert einen Prozess, bis Signal empfangen wird
       printf(„Arbeite weiter..\n");
 }
}
```

6.3.2. Modernes Signalkonzept (219)

- Versucht die obigen 3 Probleme zu vermeiden

6.4.Signale im Multithreading

- Reaktion auf Signal: Die Reaktion auf ein Signal kann von einem beliebigen Thread innerhalb eines Prozesses eingerichtet werden, gilt aber stets für den ganzen Prozess
- Signalmaske: Jeder Thread besitzteine individuelle Signalmaske
- Empfänger der Signale?
  - Synchrone Signale: Stets an den Thread, der sie ausgelöst hat
  - Asynchrones Signal: Irgendeinem Thread, der dieses nicht blockiert hat

7. Interprozesskommunikation (239)

7.1. Kommunikation

7.1.1. Synchron (242)

- Sendender und empfangender Prozess werden jeweils blockiert, bis beide Prozesse zum Meldungsaustausch bereit sind
- Keine Zwischenpufferung nötig
- Vorteile: Automatische Ablaufsynchronisation, Kein Zwischenpuffer nötig
- Nachteil: Kein unabhängiger Programmablauf möglich

7.1.2. Asynchron (242)

- Sendender und empfangender Prozess werden nicht blockiert. Sender kann einfach Senden, Empfänger holt die Meldung später ab und bestätigt den Erhalt.
- Sender und Empfänger sind entkoppelt, Zwischenspeicherung nötig
- Vorteil: Unabhängige Programmabläufe
- Nachteil: Eventuell Probleme, wenn Puffer voll. Unbestimmte Übertragungszeit

7.1.3. Halb-/ Vollduplex (244)

- Halbduplex: Nicht gleichzeitig in beide Richtungen
- Vollduplex: Gleichzeitig in beide Richtungen

7.1.4. Uni- / Bidirektional (244)

- Unidirektional: Nur in eine Richtung Datentransfer möglich
- Bidirektion: In beide Richtungen

7.1.5. Verbindungslos / Verbindungsorientiert (245)

- Verbindungsorientiert: Verbindung muss zuerst aufgebaut werden, Zuverlässige End-End-Verbindung
- Verbinungslos: Kein Verbindungsaufbau und Abbau nötig. Nachrichtenfolge nicht gewährleistet

7.1.6. Unicast / Multicast / Anycast / Broadcast (246)

- Name sagt alles..

7.2.Shared Memory

- Gemeinsam benutzter Speicherbereich.
  - Analogie: gemeinsame Tafel
- Geschwindigkeit: Schnellste Möglichkeit der IPK unter Unix
- Gegenseitiger Ausschluss beim Lesen und Schreiben von verschiedenen Prozessen notwendig. Muss vom Benutzer programmiert werden (Semaphore)
  - Analogie: Stift an der gemeinsamen Wandtafel

7.3.Pipes (247)

- Puffer für unidirektionalen Datenaustausch zwischen mehreren Prozessen
  - Analogie: Rohrpos
- Unix-Konsole: ./erzeuger | ./verbraucher
- Pipe hat Schreib- (1) und Lese-Ende (0)
- Daten werden in der Pipe gepuffert (FIFO)
- Pipes können einen Namen haben (named pipes), müssen aber nicht (unnamed pipes)
- Gegenseitiger Auschluss: Schreiben / Lesen wird nicht unterbrochen, Ausnahme: volle Pipe
- Reihenfolge des Lesens und Schreibens verschiedener Prozesse, die zum lesen / schreiben bereit sind, ist zufällig!
- Lesen aus leerer Pipe -> Prozess wird blockiert
- Schreiben in volle Pipe -> Prozess wird blockiert bis alle Daten geschrieben werden können

7.4.Sockets

- Analogie: Paketsendung, Einschreiben mit Rückantwort
- Wird oft für Kommunikation über Netzwerke verwendet
- Vielfältig und flexibler, aber komplex
- verbindungslose und verbindungsorientierte Sockets verfügbar

8. Dateisysteme (504)

Dateisysteme erlauben die dauerhafte Ablage grosser Datenmengen auf Sekundärspeichern.

8.1. Datei

- Logische Speichereinheit zur Ablage von Daten auf Sekundärspeicher
- Objekt („Datensammlung") mit Namen
- Methoden für Zugriff und Verwaltung
- Dateien werden in Verzeichnissen abgelegt (logische Organisation)
  - Unix: Alls in einem grossen Verzeichnisbaum (/usr/local/bin/)
  - Windows: Jedes Speichermedium / Partition ha einen seperaten Verzeichnisbaum (C:\...\...)
- Dateiname = Name + Dateiendung
  - Unix: Extensions nur für User und Anwendungsprogramme, jedoch keine Bedeutung fürs Dateisystem. Ausführbare Dateien haben „magische Zahl" am Dateianfang.
  - Windows: Extension .com / .exe -> Ausführbare Datei
- Dateien haben Attribute: Erzeugungsdatum, Modifikationsdatum, Besitzer, Berechtigungen, Dateityp

8.2. Blockweise Speicherung (538)

- Es wird gepuffert und blockweise auf den Blockgerätetreiber geschrieben / gelesen
- Die Blöcke werden nicht immer zusammenhängend gespeichert
- Blockgrösse in der Regel 512 Byte (CD-Rom: 2048 Byte)

8.2.1. Externe Fragmentierung (542)

- Blöcke werden nicht zusammenhängend belegt

8.2.2. Interne Fragmentierung (542)

- Blöcke sind intern nicht komplett ausgenutzt

8.3. Verwaltungsformen (543)

- Verkettete Liste
- Belegungstabelle
  - FAT (Dos / Windows), mehrfach redundant vorhanden
  - Freie Blöcke: Es wird von oben nach unten durchgegangen, wenn ein Block leer ist, dann steht in diesem der Wert -1.
- Indexlist
  - Inode (Unix)

8.4. Adressierungen (538)

- Logical Block Adressing (LBA)
  - Lesen / schreiben der Blöcke unabhängig von der Geometrie des Massenspeichers
- Zylinder-Kopf-Sektor-Adressierung (CHS)
  - Interne Adressierung von Disks, etc.

8.5. Dateizugriffe

- Wahlfrei: Auf einen bestimmten Block
- Sequenziell: Datei wird von vorne nach hinten gelesen
- Indexierter Zugriff über Schlüssel: Schlüssel muss berechnet und gespeichert werden

8.6. Partitionierung

- Mehrere Betriebssysteme mit inkompatiblen Dateisystemen
- Festplatte wird nach Benutzungszweck aufgeteilt
- Eine Festplatte ist in MBR (Master Boot Record), Partitionstabelle und verschiedene Partition aufgeteilt

8.6.1. Aufteilung Windows-Partition (550)

- Boot-Block: Grunddaten über die Platte
- FAT (File Allocation Table): Informationen über freie und belegte Blöcke, Original und Kopien
- Wurzelverzeichnis (root directory): Eine Liste aller Dateien, die in der hierarchischen Dateiverwaltung auf der obersten Ebene angelegt sind.
- Dateien: In diesem Bereich wird die Clusternummerierung angewendet (beginnt bei 2)

8.6.2. Aufteilung Unix-Partition (546)

- Boot Block
- Super Block & Verwaltungsinformationen: Info über Datenblockgrösse, Name des Dateisystems, Zeit des letzten Zugriffs
- Verwaltungsinformationen der verschiedenen Dateien (Inodes)
- Dateien und Verzeichnisse

8.6.3. Bootvorgang von Unix

- MBR (Kopf 0, Zylinder 0, Sector 1) wird beim Aufstarten des Computers vom BIOS ausgeführt, bevor zum Boot Block der aktiven Partition gesprungen wird. Dieser enthält einen Verweis auf einen bootbaren Kernel.

8.7. UFS: Unix-Dateisystem (545)

- Inodes speichern Verwaltungsinformationen in einer Datei
- Inodes enthalten: Zugriffsrechte, Typ, Grösse der Datei, Referenzzähler, Datum der letzten Inode-Änderung, des letzten Zugriffs, der letzten Modifikation, Verweise auf die Datenblöcke der Datei (Indexliste)
- Die Inode-Nummer einer Datei lässt sich mittels des Befehls „ls –i Dateiname" anzeigen.

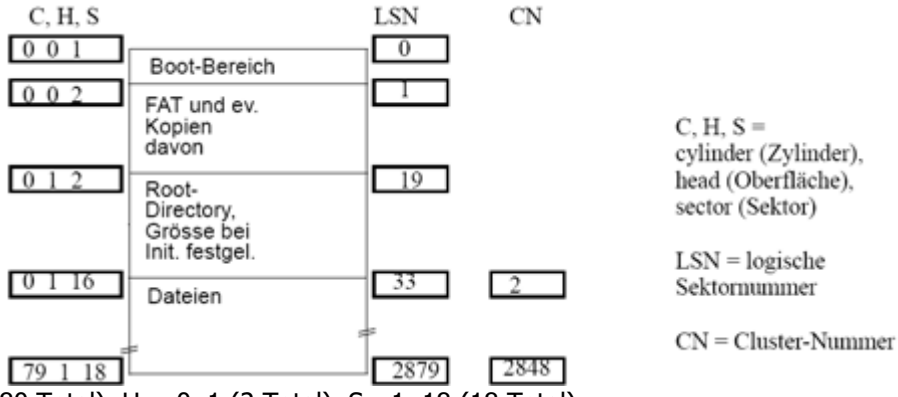
8.7.1. Hard- / Softlinks (511)

- Hardlink: Verweis auf den Inode (Name2 -> Inode -> Daten)
  - Beide Verweise haben dieselbe Inode-Nummer
  - Beim Löschen einer der beiden Inodes passiert nichts, da der Inode intern einen Referenzzähler hat. Erst wenn der Referenzzähler 0 erreicht, wird die Datei gelöscht
  - Nur innerhalb eines physischen Devices möglich
- Softlink: Verweis auf den Verzeichniseintrag (Name2 -> Inode -> Name1 -> Inode -> Daten)
  - Name2 hat eine eigene Inode-Nummer. Der Inode enthält die Pfadinformation für Name1.
  - Wird Name1 verschoben, so ist der Softlink fehlerhaft
  - Im gesamten Dateisystem möglich

8.8.FAT: Windows-Dateisystem (549)

- Partition enthält:
  - Header: Bootsektor
  - Header: FAT & Duplikat(e)
  - Header: Root-Directory (1 Verzeichniseintrag = 32Byte)
  - Dateien und Verzeichnisse
- Datei ist eine Folge von Clustern, FAT enthält das erste Cluster der Datei. Dort steht ein Verweis auf das nächste Cluster der Datei.
- FAT codiert eine verkettete Liste mit Einträgen für jedes Cluster mit Daten der Datei
- FAT12,FAT16,FAT32: XX-> Grösse eines FAT-Eintrags in Bit (12Bit bei Diskette, FAT12)
- VFAT: Lange Dateinamen unter Windows, rückwärtskompatibel zu 8+3 Namenseintrag

8.8.1. Struktur einer FAT-Partition



- C = 0..79 (80 Total), H = 0..1 (2 Total), S =1..18 (18 Total)
- Totale Clusterzahl: C*H*S = 2880
- LSN: 0..2879 (Total 2880, da 1 Sektor = 1 Cluster)
- CN: 2..2848 (= 2879 – 33 + 2)

9. CPU-Scheduling (136)

9.1. Prozesszustände (138 / 139)

- Laufend / Running
- Blockiert / Wartend / Waiting
- Bereit / Ready
- Inaktiv / Inactive

9.1.1. Prozesszustände unter Unix (168)

- Siehe Schema, zusätzliche Prozesszustände
  - Blockiert heisst in Unix sleeping / asleep
  - Laufend im User-/Kernelmode
  - Ausgelagert (swapped -> Prozesse aus dem Hauptspeicher auf den Sekundärspeicher (Disk) aus- und wieder einlagern.)
  - Zombie

9.2. Prozesstypen (143)

- CPU-lasting (CPU-bound)
- E/A-lastig (I/O-bound)

9.3. Optimierungsziele (143)

- Antwortzeit: möglichst kurze Reaktionszeit für den Benutzer
- Durchsatz: Anzahl bearbeiteter Prozess pro Zeiteinheit maximieren
- Fairness: gerechte Verteilung der CPU-Zeit
- Wartezeit: Dauer im Zustand wartend maximal ausnutzen

9.4. Strategien

- FIFO (First in, First out) -> 145
- SJF (Shortest Job First) -> 146
- SRT (Shortest Remaining Time) -> 147
- RR (Round Robin) -> 148
- ML (Multilevel Priority) -> 149
- MLF (Multilevel Feedback) ->150
  - Unix / Linux
  - Prioriätenschema, Herabsetzen von Prioritäten in Userprozessen mit dem Befehl „nice". Reduzieren der Priorität heisst „nice +19", Erhöhen „nice -20". (jeweils Maximalwerte)
  - Aging: Einmal pro Sekunde werden die Prioritäten aller Benutzerprozesse neu berechnet. In diese Berechnung fliessen der Nice-Value und die bisherige CPU-Zeit mit ein. -> 171
- RM (Rate Monotonic) -> 152
- EDF (Earliest Deadline First) -> 153

10. C-Grundlagen

10.1. Variabelnamen

- Zulässige Zeichen: Seite 10

10.2. Schlüsselwörter

- Liste Seite 10

10.3. Null-Zeichen

- Warum müssen Strings mit einem Nullzeichen (\0) terminiert werden?
  - Das Nullzeichen markiert das Ende eines Strings. Ohne dieses funktionieren verschiedene Systemaufrufe nicht einwandfrei. Beispiel: read() einer Textdatei.

10.4. Codebeispiel

```
#include <stdio.h>
long lngGlobal = 1e5; //Globale Variable, 1e5 = 100'000

intInc(int Wert); //Prototyp / Vorwärtsdekl.: Implementierung nach main()

int main(int argc, char * argv[]) {
    int arrValues[] = {1,2,3,4,5}; //Array erzeugen, int arrValues[5];
    int i=5;
    int *pi; //Zeiger auf einen INT
    char *chrTemp = "Hallo Welt!"; //chrTemp = „Adieu Welt!";
    char monat[][4] = {„I","II","III"}; //Arr mit 3 Elemente, je 4 Zeichen

    printf("%p\n",&arrValues); //Identisch mit: printf("%p",&arrValues[0]);
    printf("%.2f\n",sizeof(arrValues)); //20.00 = 5*4Byte
    pi = &i;
    printf("%i",sizeof(pi)); //4Byte = Zeigergrösse (Adressraum 32Bit)
    printf("%i\n",sizeof(*chrTemp)); //1Byte = Zeigt auf 1-Byte
}

int Inc(int Wert) {
 return(++Wert);
}
```

10.5. Arithmetische Operatoren (23)

```
int x=8,y=10,z=3,a,b,c;
b = x%z; //Modulo-Division, Rest, b=2
c = x/y; //c=0
```

10.6. Zahlenformate

```
int a=15, b=017, c=0xf; // Dezimal, Oktal, Hexadezimal, alles 15!
```

10.7. Bitoperatoren (27)

```
#include <stdio.h>
int main(int argc, char *argv[]) {
 int intVal=4, intVal2, intErg;
 intErg = intVal << 2; //100 << 2 => 10000 = 16
 intErg = intVal >> 1; //100 >> 1 => 10 = 2
 intVal = 7; //0111
 intVal2 = 11; //1011
 intErg = intVal & intVal2; //0011 = 3
 intErg = intVal | intVal2; //1111 = 15
 intErg = intVal ^ intVal2; //1100 = 12
 intErg = ~intVal2; //Zweierkomplement, -(Wert+1) = -12
}
```

10.8. Zeiger

```
//Zeigerprogramm
```

```
#include <stdio.h>

int main(int argc, char *argv[]) {

 static int intLokal;  //Erhält feste Speicheradresse!
 int intLokal2 = 10;
 int * pIntLokal;
 int * pIntLokal2;
 int ** pIntLokal3;

 scanf("%i",&intLokal); //scanf("%i %i",&intLokal1,&intLokal2);
 pIntLokal = &intLokal;
 printf("Adresse des Zeigers: %p\n",pIntLokal);
 //pIntLokal enthält als Wert die Adresse
 printf("Zeiger auf Integer: %i\n",*pIntLokal);
 //* gibt Wert unter der Adresse von pIntLokal

 pIntLokal2 = pIntLokal;
 //Adresse wird einfach kopiert
 printf("Adresse des Zeigers 2: %p\n",pIntLokal2);
 printf("Zeiger auf Integer:%i\n",*pIntLokal2);

 pIntLokal3 = &pIntLokal;
 printf("Adresse des Zeigers 3 auf Zeiger 1: %p\n",pIntLokal3);
 printf("Inhalt des Zeigers 3: %i\n",**pIntLokal3);

 return 0;
}
```

10.9. Typecast

```
void * thread_func (void * arg) {
 int * pvalue = (int *) arg; //Interpretiere als Int-Pointer
}

// void * = Zeiger auf nicht näher definierten Datentyp
```

10.10. Fallstrick Fliesskomma

```
//Beispiel 1
double a;
a = 2/3; //Gibt 0! Richtig wäre a= 2.0/3.0;

//Beispiel 2
double a = 2*0.5;
if (a == 1.0) {
 //Wird wegen Rundungsfehler nicht funktionieren!
 //if ((a > 0.9999) && (a < 1.00001))
}
```

10.11. Stolpersteine bei scanf()

- scanf() bricht bei einem Leerzeichen ab

```
char text[80];
scanf(„%s“, text); //Achtung, text ist ein Zeiger, also kein & nötig!
```

11. Linux

11.1. Wichtige Befehle

&	Program im Hintergrund starten
cat	Datei ausgeben
cd ~	Ins Homeverzeichnis (~) wechseln
chmod	Dateiberechtigungen ändern
cp	Copy
date	Aktuelle Systemzeit
echo	Ausgabe von Text oder Umgebungsvariablen
gcc	Gnu C Compiler (gcc test.c -o test)
grep	Ausgabe Filtern, cat test.txt | grep blubb -> filtert das Wort blubb
help	Siehe man
less	Datei ausgeben, Seitenweise
ls	Dateien auflisten (-al -> Alle Dateien in Listenform)
man	Manpages
mkdir	Verzeichnis erstellen
mount	Einhängen von Laufwerken
nohup ./blubb	Befehle laufen weiter, wenn von System abgemeldet wird
ps	Listes Prozesse auf
pstree	Prozesshierarchie
pwd	Aktuelles Verzeichnis
top	Welche Prozesse laufen? Systemlast?
umount	Auskoppeln von Laufwerken
whereis	Suche nach Befehl
who	Wer ist eingeloggt?
whoami	Als wer bin ich eingeloggt