LEHRSTUHL FÜR BETRIEBSSYSTEME
Univ.-Prof. Dr. habil. Thomas Bemmerl

# BETRIEBSSYSTEMPRAKTIKUM: REALZEITVERARBEITUNG

SKRIPT ZUM

# PRAKTIKUM

**Univ.-Prof. Dr. habil. Thomas Bemmerl**
**Dipl.-Inform. Stefan Lankes**
**Dipl.-Ing. Andreas Jabs**
**Dipl.-Ing. Thomas Grossmann**

Wintersemester 03/04

**Für die Hilfe und freundliche Unterstützung möchte ich mich bei folgenden Mitarbeitern bedanken:**

**Dipl.-Ing. Michael Pfeiffer**
**Dipl.-Ing. (FH) Andreas Schaaf**
**Torsten Platzbecker**
**Christian Benien**

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 Ziele und Methodik des Praktikums

In diesem Praktikum sollen Betriebssystemstrukturen und Algorithmen anhand einiger praktischer Aufgaben erlernt werden. Dabei werden hauptsächlich Realzeitsysteme, verwendet. Hierzu werden zunächst die wichtigsten theoretischen Grundlagen erläutert, die der Praktikumsteilnehmer in der Praxis anwenden soll. Es wird vorausgesetzt, dass der Praktikumsteilnehmer bereits Erfahrungen mit den Programmiersprachen C bzw. C++ hat. Vorkenntnisse mit einem UNIX-Betriebssystem (z.B.: Solaris, Linux, FreeBSD) sind hilfreich. Die Betreuer des Praktikums helfen natürlich bei Problemen mit C und UNIX, allerdings beinhaltet das Praktikum keine Einführung in C oder UNIX. Zusätzlich findet bei Bedarf nach Vereinbarung im Seminarraum (N205) des Lehrstuhls eine Besprechung statt, in der aktuelle Probleme der Teilnehmer erläutert werden.

Das Praktikum gliedert sich in vier Versuchsblöcke. Die Versuche werden von den Teilnehmern in Zweiergruppen bearbeitet. Wenn einzelne Praktikumsteilnehmer übrig bleiben, werden diese von uns zu Zweiergruppen zusammengefasst, wobei u.U. auch ein Wechsel der Gruppe notwendig wird. Neben der gemeinsamen Arbeit während der Versuchstermine an den Rechnern des lehrstuhleigenen CIP-Pools können, wenn Bedarf oder Interesse besteht, die beiden Gruppenmitglieder durchaus zu Hause oder nachmittags im CIP-Pool verschiedene Teile des Problems parallel bearbeiten. Zur Versuchsabnahme muss aber jedes Gruppenmitglied die Funktion und den Algorithmus aller Teile des Programms erklären können.

## 1.2 Grundregeln und Formalia

Wie in jedem Praktikum gibt es einige Spielregeln, die eingehalten werden müssen. Diese Regeln werden im folgenden Abschnitt genau erläutert, damit keine Missverständnisse aufkommen.

Das Praktikum findet im Raum N104 des Lehrstuhls für Betriebssysteme statt. Wie in Praktikumsräumen allgemein üblich, ist das **Essen, Trinken und Rauchen** im Raum generell **nicht gestattet**. Getränke sind immer noch die ärgsten Feinde der Tastatur! Außerdem sollte man sich immer nur so laut unterhalten, dass die Nachbargruppe nicht gestört wird. Der Raum sollte in einem sauberen und ordentlichen Zustand hinterlassen werden. Es wird gebeten, **nicht auf die Monitore zu fassen**.

Die Teilnehmer können an den vereinbarten Terminen die Versuche bearbeiten. Die Praktikumstermine sind keine anwesenheitspflichtigen Veranstaltungen. Beratung außerhalb der Termine kann jedoch nicht garantiert werden. Desweiteren ist außerhalb der Termine das Arbeiten im CIP-Pool tagsüber generell möglich, wenn dieser nicht durch andere Veranstaltungen belegt ist. Wenn durch Krankheit oder ähnliches ein Abnahmetermin nicht wahrgenommen werden kann, so sollte dies vorher mit dem Betreuer abgesprochen werden.

Kurze Fragen sollten durch eigene Versuche oder durch Fragen untereinander geklärt werden. Für tiefergehende Fragen (möglichst sammeln) steht ein Betreuer während der angegebenen Versuchszeiten zur Verfügung.

Nach den Bearbeitungsterminen eines Versuchsblocks erfolgt jeweils eine Abnahme. Die Abnahmetermine kann man auf den Webseite des Praktikums dem Zeitplan entnehmen. Um das Praktikum zu bestehen, müssen alle Versuchsblöcke erfolgreich abgeschlossen werden. Zur Abnahme eines Blocks ist die im Versuchsblock geforderte Ausarbeitung, ein Programmlisting und die Vorführung des fertigen Programms erforderlich. Außerdem müssen beide Gruppenmitglieder in der Lage sein, jeden Programmteil zu erläutern. Alle Programme sind zu kommentieren, wie es bei einem guten Programmierstil üblich ist. Verschiebung der Abnahmen ist nur in begründeten Sonderfällen in Absprache mit dem Betreuer möglich.

## 1.3 Entwicklungsplattform

Das Praktikum wird auf Sun-Rays und PCs durchgeführt, die im CIP-Pool des Instituts zur Verfügung stehen. Im Praktikum kommen die unterschiedlichsten Betriebssysteme zum Einsatz. Als Entwicklungsplattform werden die Sun-Workstations verwendet, auf denen *Solaris* installiert ist, wobei der Zugang auf die Sun-Workstations über Sun-Ray Terminals erfolgt. Die Realzeitsysteme verwenden meistens x86-basierte Architekturen. Aus diesem Grund befinden sich einige PCs im CIP-Pool des Instituts. Auf diesen sind u.a. *LynxOS, Linux, QNX* und *Real-Time Linux* installiert. Diese Systeme sollen nicht als Arbeitsplatzrechner verwendet werden, da sie nur in begrenztem Rahmen zur Verfügung stehen. Auf den Sun-Rays soll die eigentliche Entwicklungsarbeit stattfinden, während auf den PCs die Programme ausgetestet werden. Im Praktikum soll der GNU C++ Compiler verwendet werden, der auf allen Systemen zur Verfügung steht. So nützliche Werkzeuge wie der GDB, DDD und GNU Make stehen ebenfalls zur Verfügung und sollten intensiv eingesetzt werden. Eine Einführung in diese Umgebung sowie Referenzdokumentation findet man unter [11] und im Anhang E, F und G.

Zum „Einloggen“ steht jeder Gruppe eine eigene Kennung „swpg<n>“ zur Verfügung, wobei <n> für die Gruppennummer steht. Das Passwort wird den einzelnen Gruppen zugewiesen. Es sollte **nicht** verändert werden. Bei den Sun-Rays ist die grafische Oberfläche bereits gestartet und man kann sich direkt grafisch einloggen. Auf die Linux- bzw. LynxOS-Systemen wird sich über die Sun-Ray Terminals eingeloggt mit *rlogin Rechnername*.

Der Lehrstuhl besitzt den Server *zarquon* mit zwei Prozessoren (UltraSPARC-II) mit dem die Sun-Ray Terminals verbunden sind. Alle Programme, die nun in diesem Sun Ray Terminal gestartet werden, laufen auf dem Server *zarquon*. Damit die graphische Ausgabe eines anderen Rechners als *zarquon* auf dem Bildschirm des Sun-Ray Terminals erscheint, muss in diesem Terminal-Fenster noch *export DISPLAY=zarquon:Displaynummer* eingegeben werden, nachdem man sich mit *rlogin Rechnername* auf einem anderen Rechner eingeloggt hat. So wird dem Rechner mitgeteilt, dass er sein Display auf dem Sun-Ray Terminal darstellen soll. Durch das Anhängen von *&* an den Befehl läuft die Anwendung im Hintergrund weiter. Die Homepage des Internet-Browsers ist eine vorbereitete Seite (*http://www.lfbs.rwth-aachen.de/~tom/Praktikum*) mit allen Links, die für dieses Praktikum relevant sind. Darüber hinaus sind eine Reihe von Links zum Themenbereich des Praktikums und den Arbeitsgebieten des Lehrstuhls aufgeführt.

Falls Sourcecode außerhalb der Rechner des CIP-Pools geschrieben wurde, kann man sich diesen per eMail zusenden. Jede Gruppe besitzt eine eigene eMail-Adresse, die wie folgt lautet: *login_name@lfbs.rwth-aachen.de.* Über den Rechner zarquon ist eine SSH-

Verbindung von ausserhalb möglich, darüber ist auch ein Dateitransfer möglich (scp).

# 2 Echtzeitsysteme / real time systems

## 2.1 Definition des Begriffs Echtzeit

Jedes Programm, das auf einem Rechner ausgeführt wird, ist in einen zeitlichen Rahmen eingebunden. Auch wenn es sich nur in der Form darbietet, dass ein Anwender ungeduldig vor seinem Bildschirm auf die Ausgabe von Rechenergebnissen wartet. Zeitliche Unwägbarkeiten können von vielen Komponenten, die im Laufe einer Berechnung eine Rolle spielen, herrühren. Das gilt insbesondere für Zugriffe auf Platten, Speicherverwaltung, Dienste von Netzwerken, Interaktionen mit den Benutzeroberflächen, aber auch für die Ausführung der eigentlichen Berechnung. Dennoch wird in diesem Kontext nicht von einem Echtzeitsystem gesprochen.

Herausragende Eigenschaft von Echtzeitsystemen sind die von den Anwendungen **vorgegebenen** Zeitbedingungen. Eine Anwendung stellt sich als Vorgang eines technischen Systems dar, der mit Hilfe eines Rechensystems erfasst, behandelt und gesteuert werden muss. Aus Sicht der Informatik ergibt sich damit eine markante Aufteilung in ein externes System, das die anwendungsspezifischen Zeitbedingungen vorgibt, und ein internes System, das die vorgegebenen Zeitbedingungen zu beachten hat. Diese Art, ein Rechensystem zu betreiben, heißt Echtzeitbetrieb (oder Realzeitbetrieb) und ist nach DIN 44300 genormt:

> *Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.*

Somit ist ein Echtzeitsystem eine Hardware-/Softwarekombination, die Daten empfängt, diese verarbeitet und die Ergebnisse innerhalb einer definierbaren Zeitspanne an den Prozess weitergibt. Sicherlich ist diese Definition bereits etwas antiquiert und versagt unter strenger Betrachtungsweise. Dennoch wird auch hier die oben angesprochene Aufteilung in intern ablaufenden Programmen und von außen vorgegebenen Zeitspannen deutlich. Offen bleibt jedoch dabei, in welchem Maße diese Vorgaben verpflichtend sind. Wenn es die Anwendung zulässt, dass

- es genügt, die Zeitbedingungen für den überwiegenden Teil der Fälle zu erfüllen

oder

- sich geringfügige Überschreitungen der Zeitbedingungen ergeben,

dann wird von *weichen* Zeitbedingungen gesprochen. Als Beispiel für typische weiche Echtzeitanforderung soll Folgendes geleistet werden: An den Schaltern eines Reisebüros soll die Buchung eines Sitzplatzes in einem Flugzeug in 90% der Fälle weniger als 10 Sekunden und in 99% der Fälle weniger als 20 Sekunden dauern. Harte Echtzeitbedingungen zeichnen sich entsprechend dadurch aus, dass

- unter allen Umständen
- keinerlei Überschreitungen

von Zeitbedingungen auftreten. Dennoch sind auch Echtzeitsysteme nicht in der Lage, das Unmögliche möglich zu machen. So kann es zum Einen zu Ausfällen technischer Art, zum Beispiel der Rechenanlage, des Netzwerkes usw. kommen. Maßnahmen zur Fehlertoleranz können diese Problematik nur graduell verbessern. Zum Anderen kann es harte Zeitbedingungen geben, deren Erfüllung nur unter gewissen Randbedingungen sinnvoll sind.

Die Echtzeitplanung ist ein wesentlicher Bestandteil der Sicherheitsprüfung von Echtzeitsystemen. Ausgehend von einem allgemeinen Prozessmodell wird im Abschnitt 2.3 eine grundlegende Strategie zur Echtzeitplanung vorgestellt.

## 2.2 Das Prozessmodell

Im Mittelpunkt der Betrachtung stehen Rechensysteme, die über *einen* Prozessor verfügen (*Einprozessorsysteme)*. Jeder Prozess ist sequentiell auf dem Prozessor auszuführen und endet nach endlich vielen Schritten. Unter Echtzeitplanung (engl.: *real-time scheduling*) wird unter diesen Voraussetzungen die Aufgabe verstanden, den Prozessen den Prozessor so zuzuteilen, dass die von der Anwendung herrührenden Zeitbedingungen eingehalten werden. Dabei geht die Echtzeitplanung vorwiegend von harten Zeitbedingungen aus. Abhängig von der Anwendung ist zwischen Prozessen zu unterscheiden,

- deren Ausführung zwischen Start und Abschluss nicht unterbrochen werden kann - nicht verdrängbare Prozesse (engl.: *non preemptive*)
- deren Ausführung - es sei denn, es sind besondere Einschränkungen vorhanden - unterbrochen werden kann - verdrängbare Prozesse (engl.: *preemptive*).

Bei einer Prozessumschaltung (engl.: *context switch*) wird der Prozessor an einen anderen Prozess vergeben. Die Zeit für die Umschaltung findet in den nachfolgenden Planungsverfahren keine unmittelbare Berücksichtigung, auch wenn dieser Vorgang bei unterbrechbaren Prozessen sehr häufig (größenordnungsmäßig alle 10 *ms*) sein kann.

Der Start eines Prozesses, wie auch Prozessumschaltungen, können vom Rechensystem, aber auch vom technischen System ausgehen. Grundsätzlich kann ein derartiges Ereignis jederzeit geschehen. Bezogen auf den Prozessstart wird von einem *periodischen* Prozess gesprochen, falls er jeweils nach Ablauf einer festen Zeitspanne gestartet werden soll. Andernfalls heißt der Prozess *aperiodisch* oder *sporadisch*.

Wenn im Folgenden den Prozessausführungen eine Reihe fester Zeitpunkte und Zeitspannen zugeordnet werden, so stellt das fraglos eine starke Vereinfachung dar. Dennoch verlangt die einführende Betrachtung nach solchen Vereinfachungen, wenn es gilt, grundsätzliche Ereignisse bei Planungsverfahren herzuleiten. Außerdem wird sich zeigen, dass sich trotz dieser starken Vereinfachungen die Lösungen einiger Fragestellungen als widerspenstig erweisen.

Einem Prozess $P_i$ werden die folgenden Zeitpunkte bzw. Zeitspannen zugeordnet:

**Bereitzeit** (engl.: *ready time*): $r_i$
Dies ist der früheste Zeitpunkt, an dem der Prozessor dem Prozess $P_i$ zugeteilt werden darf.

**Ausführungszeit** (engl.: *execution time*): $\Delta e_i$
Diese Zeitspanne entspricht der reinen Rechenzeit auf dem Prozessor. Dabei werden für die Prozessausführung diejenigen Daten zugrunde gelegt, die die längste mögliche Rechenzeit verursachen.

**Frist** (engl.: *deadline*): $d_i$
Zu diesem Zeitpunkt muss die Ausführung des Prozesses $P_i$ beendet sein.

Die eingeführten Größen werden für die Echtzeitplanung von sporadischen Prozessen benötigt. Weitere Planungsgrößen beziehen sich auf die Ausführung von $P_i$:

**Startzeit** (engl.: *starting time*): $s_i$
Zu diesem Zeitpunkt beginnt der Prozess seine Ausführung durch den Prozessor.

**Abschlusszeit** (engl.: *completion time*): $c_i$
Zu diesem Zeitpunkt beendet der Prozess seine Ausführung durch den Prozessor.

Während die allgemeine Beziehung zwischen Start- und Abschlusszeit durch die Ungleichung

$$s_i + \Delta e_i \leq c_i$$

ausgedrückt wird, gilt für nicht unterbrechbare Prozesse die Beziehung:

$$s_i + \Delta e_i = c_i$$

Bei einem unterbrechbaren Prozess $P_i$ ist die Ausführungszeit $\Delta e_i$ über das Zeitintervall $[s_i,c_i]$ verstreut.

Für periodische Prozesse eignet sich eine leicht modifizierte Beschreibung, denn Bereitzeiten und Fristen werden von der Periode bestimmt.

**Periode** (engl.: *period*): $\Delta p_i$
Die Zeitspanne $\Delta p_i$ definiert für einen periodischen Prozess den Rahmen seiner j-ten Ausführung $P_i^j$ .

Ausgehend von seiner ersten $r_i^1$ Bereitzeit sind alle folgenden Bereitzeiten festgelegt durch:

$$r_i^j = (j-1)\Delta p_i + r_i^1 \qquad j \geq 1$$

Das Ende einer Periode ist gleichzeitig eine Frist für die Prozessausführung.

$$d_i^j = j\Delta p_i + r_i^1 \qquad j \geq 1$$

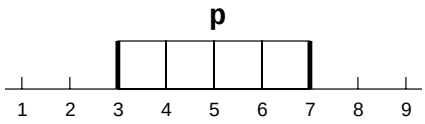
$$= r_i^{j+1}$$

Die Bereitzeiten, Ausführungszeiten, Fristen und Perioden sind als von der Anwendung vorgegebene Eingabedaten anzusehen. Der Echtzeitplanung obliegt die Festlegung der Start- und Abschlusszeiten und bei unterbrechbaren Prozessen zusätzlich die Folge von Unterbrechungsintervallen.

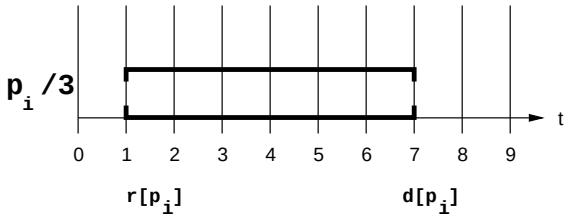
Je nachdem, ob die Prozessgrößen vor oder erst während der Ausführung bekannt sind bzw. werden, spricht man von

- statischer Planung
  Prozessdaten sind vor der Ausführung bekannt.
- dynamischer Planung
  Prozessdaten sind erst zur Laufzeit bestimmbar.

Zur Veranschaulichung werden im Folgenden verschiedene Darstellungen zur Beschreibung der Prozesse und Pläne verwendet. Ein eingeplanter Prozess mit der Startzeit $s$ = 3 und der Ausführungszeit $\Delta e = 4$ wird durch nachstehende Abbildung dargestellt.:




Ein einzuplanender Prozess mit der Bereitzeit $r_i$ = 1, Frist $d_i$ = 7 und Ausführungszeit $\Delta e_i = 3$ wird wie folgt dargestellt:




## 2.3 Planen nach Fristen

Das Planen nach Fristen (EDF = Earliest Deadline First) ist eine der verbreitesten Planungsstrategien. Das rührt weniger aus der im Einzelfall erreichten Güte eines Planes, als vielmehr aus der Breite der Einsetzbarkeit der Strategie. Planen nach Fristen lässt sich anwenden:

- auf unterbrechbare und nicht unterbrechbare Prozesse
- in statischen und dynamischen Planungsverfahren

Die Strategie besagt: Teile dem Prozessor jeweils denjenigen rechenbereiten Prozess $i$ zu, dessen Frist $d_i$ den kleinsten Wert hat. Wenn es keinen rechenbereiten Prozess gibt, dann bleibt der Prozessor untätig (engl.: *idle*).

Diese Strategie kann bei unterbrechbaren Prozessen jederzeit, d.h. entsprechend der Granularität $\Delta t_G$, zum Zuge kommen. Bei nicht unterbrechbaren Prozessen ist die Strategie nach jedem Abschluss eines Prozesses anzuwenden. Diese Strategie sei für die folgenden Verfahren in der Funktion *append($PL_k$, i)* enthalten, die aus dem bisherigen Plan $PL_k$ durch Einfügen von Prozess $i$ den Plan $PL_{k+1}$ erzeugt. Die Fristenplanung ist

optimal, wenn bei nicht unterbrechbaren Prozessen alle Bereitzeiten gleich sind.

Die Abbildung 2.1 stellt den Algorithmus dar. Dabei wird vorausgesetzt, dass die Pro-

```
EDF_schedule (PL, P)
{
  PL = <>;
  i = 1;
  while ((i ≤ n) && feasiblePL(PL,i))
{
    PL = deadlinePL(PL, i);
    i++;
  }
}
```

**Abbildung 2.1:** Algorithmus für das Planen nach Fristen von nicht unterbrechbaren Prozessen

zesse $P = \{1,\ldots,n\}$ bereits ihren Fristen nach geordnet sind: $1 \le i \le j \le n \Rightarrow d_i \le d_j$. Als Beispiel zeigt die Abbildung 2.3 die Lösung des Planungsverfahrens für das Problem aus

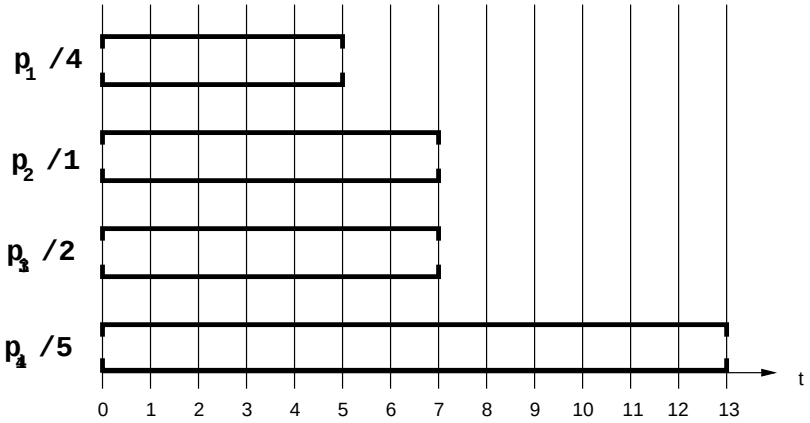



**Abbildung 2.2:** Prozessgrössen mit gleichen Bereitzeite

der Abbildung 2.2. Gänzlich unbenutzbar wird dieses Verfahren bei nicht unterbrechba-

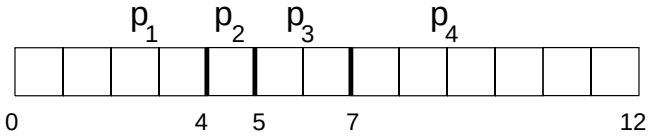



**Abbildung 2.3:** Der gefundene Plan

ren Prozessen, wenn die Bereitzeiten der Prozesse nicht gleich sind.

Die Planung nach Fristen lässt sich auf unterbrechbare Prozesse ausdehnen. Für einen Plan *PL* ist die Darstellung als Folge von Tupeln der Form $(i, l_{i,j}, \Delta l_{i,j})$ geeignet. Die Ausführungszeit eines Prozesses *i* ist im Plan *PL* in $m_i$ Zeitintervalle zerlegt. Ein Intervall beginnt bei $l_{i,j}$ und dauert $\Delta l_{i,j}$ Zeiteinheiten. Für einen brauchbaren Plan gelten fol-

gende Randbedingungen:

$$\sum \Delta l_{i,j} = \Delta e_i$$

und

$$r_i \le s_i = l_{i,1} \le l_{i,m_i} + \Delta l_{i,m_i} = c_i \le d_i$$

Die Strategie der Planung nach Fristen kann zu einem Zeitpunkt *t* nur auf denjenigen Prozessen angewendet werden, die rechenbereit sind, d. h. wenn *t* zwischen Bereitzeit und Frist liegt und der Prozess noch nicht vollständig ausgeführt ist:

$$\text{Ready}(t) = \{i \mid (r_i \le t < d_i \wedge \text{rest}(i,t) > 0)\}$$

Dabei umfasst *rest(i,t)* alle Ausführungszeiten von Prozess *i*, die in den bis zum Zeitpunkt *t* ausgebauten Plan *PL* noch nicht aufgenommen wurden:

$$\text{rest}(i,t) = \Delta e_i - \sum_{j \in PL} \Delta l_{i,j}$$

Zum Zeitpunkt *t* soll derjenige Prozess aus *Ready(t)* rechnen, dessen Frist die kürzeste ist (engl.: *earliest deadline first*). Mit der Funktion *edf(Ready(t))* wird ein solcher Prozess bestimmt. Es bleibt zu klären, welches Zeitintervall dem Prozess ab *t* zugeordnet werden kann. Zum einen kann es nicht mehr sein, als er noch Ausführungszeit hat: *rest(i,t)*. Zum anderen kann es, vom Zeitpunkt *t* aus gesehen, einen nächsten Zeitpunkt geben, zu dem weitere, bislang unberücksichtigte Prozesse mit ihren jeweiligen Fristen betrachtet werden müssen. Dieser Zeitpunkt berechnet sich so:

$$\text{nextavail}(t) = \begin{cases} \min\{r_i \mid r_i > t\}, & \text{falls } \{r_i \mid r_i > t\} \neq \emptyset \\ \max\{d_i\}, & \text{sonst} \end{cases}$$

Den oben unterschiedenen Fällen gemäß wird der Bezugszeitpunkt *t* entweder auf *t+rest(i,t)* oder auf *nextavail(t)* erhöht, um dann im Planungsverfahren fortzufahren.

Das Planungsverfahren endet, wenn zu einem Zeitpunkt *t* die Ausführungszeiten aller Prozesse im Plan *PL* enthalten sind. Zu diesem Zweck liefert die boolesche Funktion *allinPL(t)* genau dann den Wert Wahr, wenn gilt:

$$\sum_{i \in P} \text{rest}(i,t) = 0$$

Das Planungsverfahren kann aber schon früher enden, und zwar dann, wenn ein rechenbereiter Prozess *i* seine Frist verletzen wird:

$$\neg \text{feasiblePL}(i,t) \Leftrightarrow t + \text{rest}(i,t) > d_i$$

Gibt es zu einem Zeitpunkt *t* gerade keine rechenbereiten Prozesse, dann sind entweder

schon alle Prozesse brauchbar eingeplant oder es gibt noch Prozesse mit Startzeiten, die bislang nicht berücksichtigt wurden. Sind alle Prozesse eingeplant, wird dies vom Verfahren in der Abbildung 2.4 durch die Auswertung von *allinPL(t)* erkannt. Andernfalls

```
EDF_schedule (PL, P)
{
  PL = <>;
  t = min{r_i | (r_i ∈ P)};
  while (¬ allinPL(t))
  {
    if (Ready(t) = ∅)
      t = nextavail(t);
    else {
      i = edf(Ready(t));
      if (¬ feasible(i,t)
        break;
      Δl = min(rest(i, t), nextavail(t) – t);
      PL = PL ∪ (i, t, Δl);
      t = t + Δl;
    }
}
```

**Abbildung 2.4:** Algorithmus für das Planen nach Fristen von unterbrechbaren Prozessen

erfolgt ein Vorrücken auf den Zeitpunkt *nextavail(t)*.

Wird der Algorithmus auf das Beispiel in Abbildung 2.5 angewendet, so erhält man als

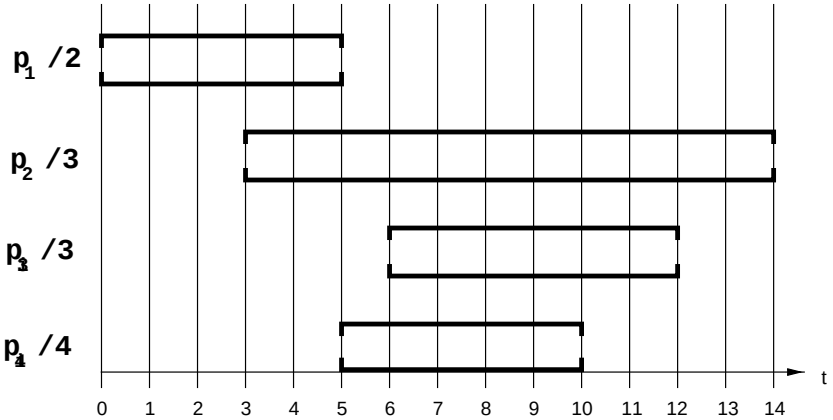



**Abbildung 2.5:** Beispiel für das Planen nach Fristen

Ergebnis einen Plan, der in Abbildung 2.6 dargestellt wird.

Das Planen nach Fristen ist für unterbrechbare Prozesse optimal. Es wird auch in dynamischen Planungsverfahren eingesetzt. Voraussetzung ist, dass Bereitzeit, Ausführungs-

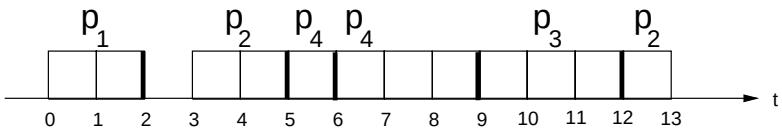



**Abbildung 2.6:** Lösung des Beispiels

zeit und Frist eines Prozesses so früh bekannt werden, dass ein einzelner Planungsschritt für die Ermittlung des nächsten Tupels erfolgen kann. Vielfach wird dabei unterstellt, dass der Zeitaufwand für die Echtzeitplanung und Prozessorzuteilung vernachlässigt werden kann.

Dynamische Fristenplanung bleibt optimal, auch wenn die Ausführungszeiten nicht bekannt sind. Ein statisches Planungsverfahren würde auch bei im Nachhinein vollständig vorliegenden Daten keinen brauchbaren Plan finden, wenn ein dynamisches Planungsverfahren keinen findet. Dies ist in Abbildung 2.4 insofern ersichtlich, dass für die Auswahl des nächsten Prozesses, d.h. in der Funktion *edf*, die Ausführungszeit unbeachtet bleibt.

# 3 Realzeitbetriebssysteme

## 3.1 Marktübersicht

Im Server und Workstation-Bereich gab es in den letzten Jahren eine Marktkonzentration. Man kann sagen, dass sich zwei Lager gebildet haben. Zum einen handelt es sich um das Windows-Lager (Windows 9x, Windows NT, Windows 2000), zum anderen um das große Lager aller UNIX-Derivate (Solaris, AIX, Linux). Der Markt für Realzeitbetriebssysteme (kurz: RTOS) ist ziemlich zersplittert. Die Zeitung *Real-Time Magazine* hat ihre Leser gefragt, welche Realzeitbetriebssysteme sie verwenden bzw. in der nächsten Zeit verwenden werden. Die Ergebnisse kann man in [20] finden und sind in Abbildung 3.1 grob zusammengefasst. Natürlich stellt diese Umfrage keine repräsentative Marktun-

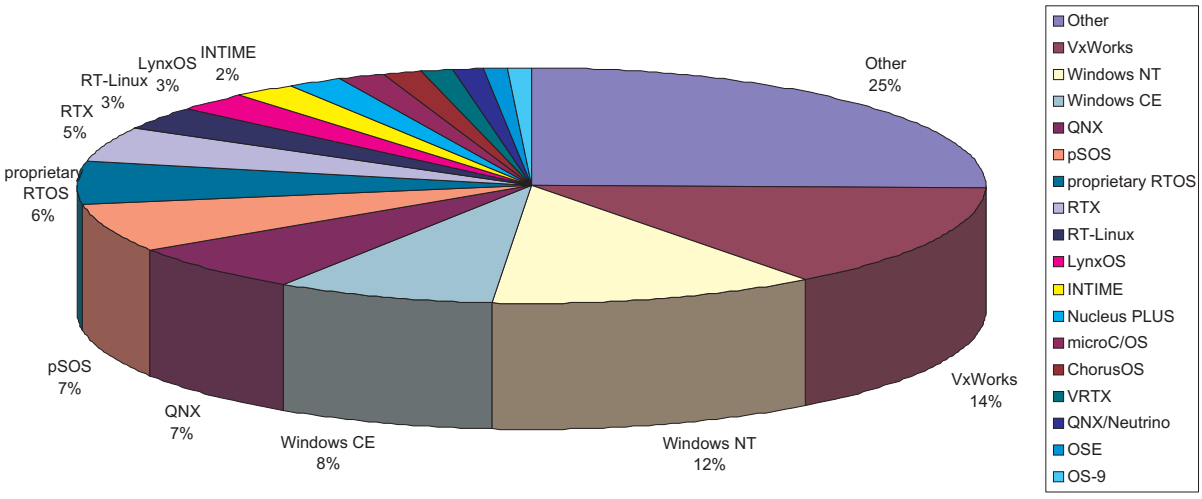



**Abbildung 3.1:** Eine nicht repräsentative Übersicht des Realzeit-Markts

tersuchung dar, aber man erhält hierdurch eine gute Marktübersicht. Man sieht sehr deutlich, dass der Markt sehr zersplittert ist. Allein 25 Prozent der Befragten gaben entweder kein Betriebssystem an oder verwenden ein Betriebssystem, das weniger als ein Prozent Marktanteil besitzt. Außerdem benutzen immer noch 6 Prozent der Firmen, die ein Realzeitbetriebssystem benötigen, ein Betriebssystem, das sie selber entwickelt haben. Diese selbst entwickelten Betriebssysteme findet man in Abbildung 3.1 unter dem Namen *„proprietary RTOS“*. Die Firmen, die Realzeitbetriebssysteme verwenden, wollen sich nicht in tausend verschiedene Betriebssysteme einarbeiten, da hierdurch die Entwicklungskosten sehr hoch werden. Sie würden lieber einen Standard vorziehen. Aus diesem Grunde wurde zum Beispiel der POSIX.4-Standard entwickelt, der genauer im Abschnitt 3 beschrieben wird.

Einige Firmen halten das Windows-API für den Defakto-Standard (Industrie-Standard) der Zukunft. Sie meinen, Windows wird sich im Bereich der Realzeitbetriebssysteme genauso durchsetzen, wie es sich bereits im Workstation-Bereich durchgesetzt hat. Deshalb verwenden sie Windows NT, da es einige Realzeiteigenschaften besitzt. Doch Windows NT ist nicht speziell für die Realzeitverarbeitung konzipiert worden und ist für eingebettete Systeme, welche kaum Ressourcen besitzen, vollkommen ungeeignet. Somit kann man Windows NT nicht in Branchen einsetzen, die harte Zeitbedingungen einhalten müssen. Windows CE hat sich bis jetzt nicht im Gebiet der Realzeitverarbeitung durchgesetzt, da es kaum Realzeiteigenschaften besitzt. Microsoft hat aber bereits angekündigt, dass man mit Windows CE 3.0 Anwendungen entwickeln kann, die harte Echtzeitanforderungen einhalten. Aus diesem Grund wollen viele Firmen auf Windows CE umsteigen. Dies erklärt den hohen Anteil von 8 Prozent in Abbildung 3.1. Auch andere Firmen haben den Windows-Trend erkannt. Zum Beispiel vertreibt die Firma QNX ein Realzeitbetriebssystem, das ebenfalls QNX heißt und eigentlich ein UNIX-basiertes Betriebssystem darstellt. QNX hat aber eine Bibliothek entwickelt, mit deren Hilfe man Programme, die unter Windows geschrieben wurden, für QNX compilieren kann. Auf diesem Weg erhält man so eine Quell-Kompatibilität zwischen den beiden so verschiedenen Betriebssystemen.

Einige andere Firmen versuchen Windows NT zu erweitern, so dass es realzeitfähig ist. Typische Vertreter dieser Betriebssystem-Erweiterungen stellen *INTIME* und *Hyperkernel* dar. Zum Beispiel stellt *Hyperkernel* ein eigenständiges Realzeit-Betriebssystem dar, in dem Windows NT als Task mit niedrigster Priorität läuft. Programme, die keinen Echtzeit-Anforderungen unterliegen, können so als normale Windows-Applikation laufen, während Programme, die gezwungen sind, harte Echtzeit-Bedingungen einzuhalten, als eigenständiger Hyperkernel-Task laufen. Durch dieses Konzept können alle Windows-Dienste verwendet werden. Dies spart Entwicklungszeit und somit auch Kosten. Ein ähnliches Konzept verfolgt *RTLinux*, das im Abschnitt 4 genauer beschrieben wird. Hierbei verwaltet allerdings *RTLinux* das Betriebssystem *Linux* als eigenständigen Task mit niedrigster Priorität.

Es stellt sich aber die Frage, ob überhaupt eine ultimative Lösung für Realzeit-Systeme existiert. Immerhin sind die Anforderungen zwischen den Realzeitsystemen äußerst unterschiedlich. Ein Server, der einen kontinuierlichen Video-Stream liefern soll, muss ganz andere Bedingung erfüllen, als der Steuerungs-Computer in einer Boing 747. Man kann sich (noch) kaum vorstellen, dass Windows in einer Boing 747 eingesetzt wird. Es steht aber fest, dass eine Marktkonzentration stattfinden wird. Zur Zeit sind *VxWorks, QNX, pSOS* und *LynxOS* wohl die bekanntesten kommerziellen Betriebssysteme, die harte Echtzeit-Bedingungen einhalten. In diesem Praktikum wird näher auf *RTLinux*, *QNX* und *LynxOS* eingegangen.

## 3.2 Eigenschaften eines Realzeitbetriebssystems

Im Bereich der Realzeitsysteme gibt es immer noch viele Entwicklungen, die ohne den Einsatz von Betriebssystemen auf dem Zielsystem auskommen. Das trifft in besonderem Maße für eingebettete Systeme zu, sofern sie ein enges Aufgabenfeld abdecken und nur einfache Ein- und Ausgabeoperationen in Anspruch nehmen.

Für Realzeit-Anwendungen sind die meisten Betriebssysteme (Windows, Solaris, Linux,...) nur sehr eingeschränkt verwendbar. Zum einen sind Architekturen der ver-

schiedenen Kerne für Echtzeitverarbeitung nicht geeignet (Job-Fairness statt feste Prioritätsverteilung, nicht-unterbrechbare, langsame Interruptverarbeitung etc.), zum anderen macht die schiere Größe des Betriebssystems den Einsatz in eingebetteten Systemen ohne Festplatte fast unmöglich.

Die Beantwortung der Frage, was Realzeitbetriebssysteme von anderen Betriebssystemen abhebt, orientiert sich an den ureigenen Forderungen für Echzeitsysteme:

- Formulierung und Einhaltung von Zeitbedingungen
- Vorhersagbarkeit des Systemverhaltens

Aus diesen grundlegenden Forderungen leiten sich, bezogen auf die verschiedenen Komponenten von Betriebssystemen, eine Reihe von wünschenswerten Merkmalen ab:

- Unterbrechungen von unterschiedlicher Dringlichkeit sind vom Betriebssystem aufzufangen und gegebenenfalls an die Anwenderprogramme weiterzuleiten.
- Bezogen auf die Einplanung von Prozessen soll der Benutzer mindestens imstande sein, den Prozessen anwendungsspezifische Prioritäten zuzuordnen.
- Dem Benutzer sollen zeitabhängige Operationen, zum Beispiel zum Wecken von Prozessen, zur Verfügung stehen. Die Granularität des zugrunde liegenden Zeitrasters sollte hinreichend fein sein.
- Der Kern eines Betriebssystems sollte unterbrechbar sein, damit die Ausführung eines höher prioritisierten Prozesses allzeit Vorrang vor der eines niedriger prioritisierten hat.
- Es sollte die Möglichkeit geben, Ein-/Ausgabeoperationen asynchron abzuwickeln um so zu vermeiden, dass ein Prozess untätig auf das Ende einer solchen Operation warten muss.
- Unwägbarkeiten beim Zugriff auf Dateien, die sich auf einer Festplatte befinden, sollten dadurch ausgeschlossen werden, dass auf die Abarbeitungsreihenfolge durch den Festplattentreiber eingewirkt werden kann und durch die Organisation der Daten auf der Festplatte, zum Beispiel durch zusammenhängende Dateien, keine unvorhersehbaren Zugriffszeiten auf die ansonsten über die Platte verstreuten Datenblöcke entstehen.
- Eine virtuelle Speicherverwaltung sollte den Anwendungsprozessen die Möglichkeiten bieten, sich selbst von unvermittelten Seitenwechseln (engl. *paging*) auszuschließen, da es ansonsten zu erheblichen und unvorhersehbaren Verzögerungen in der Prozessausführung kommt.
- Die Anbindung an das Rechnernetz sollte vom Betriebssystem unterstützt werden und auf Kommunikationsprotokolle aufsetzen, die die Vorhersagbarkeit bezüglich Dauer und Reihenfolge der zu übertragenden Nachrichten sicherstellen.
- Realzeitbetriebssysteme sollten konfigurierbar sein, damit man das Betriebssystem optimal an die gegebene Plattform anpassen kann. Dadurch erhält man ein res-

sourcenschonendes Betriebssystem. Diese Eigenschaft wird von eingebetteten Systemen verlangt.

Aus den obigen Punkten kann man aus technischer Sicht die folgenden Aspekte zusammenstellen:

- minimale Antwortzeiten bei hoher Systemlast
- geringe Interrupt-Latenzzeiten
- geringe Kosten für einen Kontextwechsel
- unterbrechbares prioritäten-gesteuertes Prozessmodel
- Vermeidung von Prioritäten-Inversionen
- geringe Kosten für einen Kern- bzw. Treiber-Eintritt
- das Betriebssystem sollte konfigurierbar sein
- schlankes System, d.h. das System benötigt wenige Ressourcen

Viele Betriebssysteme, die technisch äußerst innovativ waren, konnten sich auf dem Markt nicht durchsetzen. Um sich auf den Markt durchzusetzen, benötigt ein Betriebssystem noch einige andere Eigenschaften. Diese kann man kaum spezifizieren. Die folgende, subjektiv erstellte Liste versucht aber, solche Eigenschaften zu beschreiben:

- die Standard-Dienste und -Treiber (TCP/IP, Ethernet, SCSI, Serielle-, Parallele-Schnittstelle) sollten zur Verfügung stehen
- es sollte leicht zu erlernen sein
- es sollte eine gute Entwicklungsumgebung besitzen
- gutes Preis-Leistungsverhältnis
- gutes Marketing
- eine bereits bestehende Lobby

# 4 Real-Time Linux

Soft- und Hardwareentwicklungen sind heute dem Zwang der Wiederverwendbarkeit unterworfen. Dies gilt auch - und insbesondere - für Echtzeitaufgaben. Das Kochrezept hierfür könnte etwa so lauten: Man nehme elektronische Komponenten von der Stange - hier scheint die Industrie den PC als Plattform mehr und mehr zu entdecken, obwohl es für dieses Aufgabenfeld deutlich bessere Architekturen gäbe (der Preis macht es eben aus). Des Weiteren nehme man ein Betriebssystem von der Stange - spätestens hier hat man ein kleines Problem: Echtzeit-Betriebssysteme wie QNX, VxWorks oder pSOS+ entstammen dem Markt für eingebettete Systeme und sind zwar für die harte Zeitanforderung gut gerüstet, oft gibt es aber für die immer häufiger geforderte graphische Benutzerschnittstelle keine vernünftigen Bibliotheken oder diese entsprechen keiner der verbreiteten Normen. Im Gegensatz dazu kommen Unix-Betriebssysteme aus dem Multiuser/Multitasking-Umfeld, deren primäre Aufgabe die optimale Verteilung von Rechenzeit auf lang laufende Prozesse zum Ziel hat - kein guter Ansatz für Echtzeitaufgaben.

Selbst ein Mikrocontroller für 3 DM kann bei entsprechender Programmierung Echtzeitbedingungen besser einhalten, als ein Pentium II samt modernem „Standardbetriebssystem“, obwohl hier z.T. die tausendfache Leistung zur Verfügung stehen würde! Denn leider haben diese Betriebssysteme (streng betrachtet) keine Echtzeitfähigkeiten, wobei das schlechte Design der PC-Architektur ihr Übriges tut.

Doch speziell für Linux gibt es Abhilfe in Form von RTAI und Real-Time Linux (kurz: RT Linux).

## 4.1 Grundlagen von RT-Linux

RT Linux ist eine Echtzeiterweiterung für Linux. Hierbei werden echtzeitfähige Threads von einem (von Linux unabhängigen) erweiterbaren Minikernel verwaltet. Der Minikernel von RT-Linux und die echtzeitfähigen Threads werden wie Treiber (Kernel-Module) zum Betriebssystem nachgeladen. Somit arbeitet RT Linux auf Treiber-Ebene und kann direkt auf die Hardware zugreifen. Der eigentliche Linux-Betriebssystemkern ist aus Sicht dieses Minikernels ebenfalls nur ein Thread, allerdings mit der niedrigsten Priorität im System (Idle-Thread). Durch diese Architektur gewinnt eine Echtzeitanwendung einerseits vollständige Kontrolle über die Hardware, kann aber andererseits nur sehr bedingt Linux-Systemaufrufe nutzen. Neben preemptiven Priority-Scheduling sind auch Module für Earliest-Deadline-First-Scheduling und Kommunikationsmechanismen zwischen RT-Threads über Queues und Semaphoren erhältlich. Der Datenaustausch zwischen RT-Threads und Applikationen, die im normalen Adressraum von Linux laufen, erfolgt für gewöhnlich über spezielle RT-FIFOs. Die Philosophie der Entwickler von RT Linux lautet: so wenig wie möglich in den bestehenden Kern eingreifen, um dessen Stabilität nicht zu gefährden. Leider ist der Kernzusatz nur teilweise konform mit dem POSIX-Standard 1003.1b. Dieser Standard beschreibt Betriebssystemschnittstellen für Realzeit-Anwendungen und ist besser als POSIX.4 bekannt und wurde im Kapitel 9 ausführlich erläutert. Weitere Funktionalitäten können jederzeit mit Hilfe des Kernelmodul-Konzepts nachgeladen werden. Typische Aufgaben für RT Linux finden sich im Bereich der Mess-, Steuer- und Regelungstechnik, wo harte Deadlines unter 50 Mikrosekunden gefordert sind.

RT Linux findet bei vielen kleinen, ehrgeizigen und privaten Projekten (z.B. EYCar) sei-

nen Einsatz, wird aber auch bereits auf Tauglichkeit für teure CNC-Werkzeugmaschinen getestet (EMC-Projekt). Auch die NASA setzt in einem ihrer Versuchsflugzeuge auf RT Linux (zur Flugradarsteuerung). Interessanterweise verwenden kommerzielle, proprietäre Echtzeiterweiterungen für Windows NT dieselbe Architektur wie RT-Linux. Allerdings führt der fehlende Quellcode oft zu Wehklagen von Seiten der Softwareentwickler. Mehr Infos zu RT-Linux finden sich auf dessen Homepage unter http://www.rtlinux.org.

## 4.2 Thread auf Kernebene

Die Aufrufe zur Thread-Verwaltung sind in der Datei *rtl_sched.h* definiert. Alle zu RT Linux zugehörigen Header befinden sich im Verzeichnis */usr/src/rtlinux-2.0/rtl/include*. Die Entwickler von RT Linux haben versucht, ihre Schnittstelle an den PThread-Standard (siehe Abschnitt 8) anzulehnen, um die Entwicklung von Programmen unter RT Linux zu vereinfachen. Es sollte aber den Anwendern von RT Linux bewusst sein, dass sie auf der Ebene des Kerns arbeiten und nicht im Adressraum eines normalen Programmes.

Die meisten Befehle werden im Abschnitt 8 vorgestellt. Aus diesem Grund werden viele PThread-Befehle hier nicht näher erläutert. Die wohl wichtigste Funktion ist die Erzeugung eines echtzeitfähigen Threads:

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

Dieser Aufruf erzeugt einen neuen Thread, der seine Ausführung mit der Funktion *fn* startet. Mit Hilfe des Arguments *attr* können einige Attribute des erzeugten Threads festgelegt werden. Zum Beispiel kann so die Größe des Stacks und die Priorität des Threads bestimmt werden. Dies wird im Abschnitt 8 ausführlich erläutert.

Periodische Prozesse spielen in der Realzeitverarbeitung eine wichtige Rolle. Aus diesem Grund kann ein Thread in RT Linux als periodisch definiert werden. Dies bedeutet, dass der Thread periodisch seine Arbeit wieder aufnimmt. Hierzu muss bei der Definition eines periodischen Threads die Periodendauer und der Startzeitpunkt (um Phasenverschiebungen zwischen Threads zu ermöglichen) bestimmt werden. Dies wird durch den folgenden Befehl erreicht:

```
int pthread_make_periodic_np(pthread_t thread, hrtime_t start_time,
                             hrtime_t period);
```

Periodische Threads werden in dem PThread-Standard nicht definiert. Aus diesem Grund wurde der Befehl mit der Endung *_np*, wie alle anderen RT Linux-Befehle, die nicht zum Standard konform sind, gekennzeichnet. In diesem Zusammenhang steht die Endung *_np* für *non-portable Real-Time Linux extension*. Der Startzeitpunkt sowie die Periodenlänge werden durch die Argumente *start_time* und *period* bestimmt. Beide Argumente werden in Nanosekunden angegeben. Soll der Thread sofort gestartet werden, so wird die aktuelle Zeit als Startzeitpunkt gewählt. Dieser Zeitpunkt kann durch die Funktion *gethrtime()* ermittelt werden. Die Periodenlänge kann nicht nur durch die Funktion *pthread_make_periodic_np* angegeben werden, sondern auch nachträglich verändert werden, indem einfach der Wert der Variablen *period* in der Struktur *pthread_t* des Threads verändert wird. Dabei wird aber vorausgesetzt, dass der Thread bereits durch *pthread_make_periodic_t* als periodisch definiert ist.

Am Ende eines Bearbeitungs-Zykluses muss die CPU wieder für andere Aufgaben freigegeben werden; dies wird durch die folgende Funktion erreicht, da ein Thread beim Aufruf der Funktion blockiert, bis die Periode wieder beginnt:

```
int pthread_wait_np(void);
```

Soll die Ausführung eines Threads unterbunden werden, so wird dies mit dem folgendem Aufruf erreicht:

```
int pthread_suspend_np(pthread_t thread);
```

Die Aufhebung der Suspendierung geschieht schließlich mit der Funktion

```
int pthread_wakeup_np(pthread_t thread);
```

Ein nicht mehr gebrauchter Thread kann durch die folgende Funktion gelöscht werden:

```
int pthread_delete_np(pthread_t thread);
```

## 4.3 FIFO-Handling

Nur eine sehr kleine Klasse von echtzeitfähigen Threads kommuniziert ausschließlich mit der physikalischen Außenwelt. Der weit größere Teil der Aufgaben bedarf Interaktionen mit einem Anwendungsprogramm (z.B. GUI für die Eingabe des Sollwertes eines digitalen Reglers). Für diesen Zweck gibt es die vier Echtzeit-FIFOs (/dev/rtf0 - /dev/rtf3). Die Definition der FIFO-Funktionen befindet sich im Header *rtf_fifo.h*.

```
int rtf_create(unsigned int fifo, int size);
```

Dieser Aufruf belegt den FIFO-Kanal mit der Nummer *fifo* und der Größe *size*. Entsprechend kann man eine FIFO auch wieder freigeben:

```
int rtf_destroy(unsigned int fifo);
```

Der folgende Aufruf ermöglicht einem Thread, etwas in die FIFO zu schreiben:

```
int rtf_put(unsigned int fifo, char* buf, int count);
```

An der Adresse *buf* befinden sich Daten von insgeamt *count* Bytes, die in den FIFO-Kanal mit der Nummer *fifo* geschrieben werden. Die Funktion liefert als Ergebnis ist -1, falls die FIFO voll ist und somit nicht in die FIFO geschrieben werden kann. Entsprechend wird mit dem Befehl

```
int rtf_get(unsigned int fifo, char* buf, int count);
```

*count* Bytes Daten von einer FIFO ausgelesen und an die Adresse *buf* kopiert.. Die Funktion liefert als Ergebnis ist -1, falls die FIFO leer ist. Threads können mit Hilfe von FIFOs auch miteinander kommunizieren. Meistens wird dies aber nicht benötigt, da Threads im gemeinsamen Adressraum arbeiten und somit über den gemeinsamen Speicher kommunizieren können. Unter RT Linux kann ein FIFO-Handler definiert werden, der jedesmal aktiv wird, wenn eine Applikation etwas in ein entsprechendes Device (/dev/rtf[0-3]) hineinschreibt. Ein Programm kann unter RT Linux mit Hilfe des Befehls

```
int rtf_create_handler( unsigned int fifo,
```

```
                        int (*handler)(unsigned int fifo));
```

einen solchen FIFO-Handler definieren. Der Handler bekommt auch mitgeteilt, welches FIFO-Device von der Applikation angesprochen wurde. Somit können mehrere FIFOs von einem einzigen Handler bedient werden.

## 4.4 Zeit-Verwaltung

Im Header *rtl_time.h* wird der Datentyp *hrtime_t* definiert. Dabei handelt es sich hierbei um eine 64 Bit lange Zahl, die die Zeit im Nanosekundenbereich darstellt. Mit Hilfe der Funktion

```
hrtime_t gethrtime(void);
```

kann ermittelt werden, wieviele Nanosekunden seit dem letzten „Booten“ vergangen sind.

## 4.5 Interrupt-Handling

Mit den beiden folgenden Funktionen kann ein Interrupt an einen bestimmten Interrupt-Handler gebunden bzw. wieder freigegeben werden:

```
int rtl_request_irq(unsigned int irq,
                    unsigned int  (*handler)(unsigned int irq,
                    struct pt_regs *regs));
int rtl_free_irq(unsigned int irq);
```

wobei *irq* die verwendete Interruptnummer (0-15) und *handler* der Verweis (Funktionszeiger) auf die Interrupt-Service-Routine darstellt. Die Interrupt-Service-Routine bekommt neben der Nummer des Interrupts, der ausgelöst wurde, um die Interrupt-Service-Routine anzuspringen, auch einen Zeiger auf die Datenstruktur *struct pt_regs* übergeben. In diesem Praktikum wird diese Datenstruktur nicht benötigt. Eine Erläuterung dieser Datenstruktur wird ebenfalls nicht gegeben, da dies den Umfang des Skriptes sprengen würde. Bevor der Interrupt-Handler verlassen wird, muss die Funktion

```
void rtl_hard_enable_irq(unsigned int irq);
```

im Interrupt-Handler aufgerufen werden. Damit wird dem System mitgeteilt, dass der Interrupt bearbeitet ist und der Handler bereit ist, neue Interrupts zu behandeln.

## 4.6 Extrawürste

Da der RT-Linuxkern ein eigenes Betriebssystem darstellt, dürfen die „echten“ Linuxsystemaufrufe nur während der Initialisierungs- und der Cleanup-Phase der Realzeitanwendung Verwendung finden, da sich hier der Prozessor noch im „normalen“ Kernelmodus befindet. Bibliotheken, die keine Systemcalls beinhalten, dürfen aber **statisch** hinzugebunden werden. Die mathematische Bibliothek ist beispielsweise ein solcher Kandidat. Werden in einem Thread generell Fließkommavariablen und Fließkomma-Operationen verwendet, dann muss dies mit der Funktion

```
int pthread_setfp_np(pthread_t thread, int flag);
```

bekannt gegeben werden. flag=1 bewirkt, dass bei jedem Kontextwechsel auch die Fließkomma-Register gerettet bzw. wieder hergestellt werden. Benötigt der Thread die Fließkommaeinheit für längere Zeit nicht mehr, so kann mit *pthread_setfp_np(threadId, 0)* Rechenzeit beim Kontextwechsel gespart werden. Standardmäßig wird angenommen, dass nur Integer-Operationen verwendet werden. Weitere Befehle werden nicht benötigt, um echtzeitfähige Programme unter RT Linux zu erstellen. Es gäbe zwar noch etliche weitere Kommandos, z.B. für Queues, Semaphore, Shared Memory und Deadline-Scheduling, aber diese sind noch nicht im Standard-RT-Kern integriert und müssen separat vom Netz geholt werden. Die frisch erworbenen Kenntnisse sollen nun anhand eines praktischen Beispiels gefestigt werden.

## 4.7 Das Beispielprogramm „rectangle“

Als nächstes sollen anhand eines kleinen Beispielprogrammes (Abbildung 4.1) die einzelnen Funktionen von RT-Linux näher erläutert werden. Dieses Beispielprogramm soll ein Rechtecksignal am Parallelport ausgeben. Wie bereits erwähnt, werden Programme unter RT Linux als Treibermodule bzw. Kernelmodule geladen. Solche Module besitzen stets die Funktionen *init_module* und *cleanup_mudule.* Die Funktion *init_module* wird beim Initialisieren des Moduls aufgerufen, während beim Entfernen des Moduls *cleanup_module* aufgerufen wird. In der Funktion *init_module* werden zwei periodische Threads erzeugt. Die Periodenlänge beträgt bei beiden Threads 500 µs, doch der Startzeitpunkt ist um 200 µs voneinander versetzt. Beide Threads führen die Funktion *fun* aus, die durch die Funktion *outb* den Parameter *t* auf dem Parallelport ausgibt. Anschließend legt sich der Threads schlafen, bis er beim Beginn einer neuen Periode wieder geweckt wird und wieder auf dem Parallelport den Parameter *t* ausgibt. Ein Thread setzt das Ausgabesignal des Parallelports auf *0xffff*, während der andere Thread es wieder auf *0* zurücksetzt. Da der Startpunkt der beiden Threads um 200 µs versetzt ist, wird 200 µs *0xffff* auf dem Parallelport zu sehen sein. Da diese Ausgabe alle 500 µs wieder zu sehen ist, erscheint auf dem Parallelport ein periodisch wiederkehrendes Rechtecksymbol.

Beim Entfernen des Modules wird die Funktion *cleanup_module* aufgerufen. In dieser Funktion werden die gestarteten Threads wieder entfernt.

```
/* Produce a rectangular wave on output 0 of a parallel port */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/cons.h>
#include <asm/io.h>

#include <rtl_sched.h>

#define LPT_PORT   0x378

pthread_t mythread1;
pthread_t mythread2;
```

```
void* fun(void* arg)
{
    int t = (int) arg;

    while(1)
    {
        outb(t, LPT_PORT); /* write on the parallel port */
        pthread_wait_np();
    }

    return 0;

}

int init_module(void)
{
    hrtime_t now = gethrtime();

    /* this thread will be setting the bit */
    pthread_create(&mythread1, NULL, fun, (void*) 0xffff);

    /* this thread will be resetting the bit */
    pthread_create(&mythread2, NULL, fun, (void*) 0);

    /* the threads run periodically with an offset of 200000 ns */
    pthread_make_periodic_np(mythread1, now + 3000000, 500000);
    pthread_make_periodic_np(mythread2, now + 3200000, 500000);

    return 0;
}

void cleanup_module(void)
{
    pthread_delete_np(mythread1);
    pthread_delete_np(mythread2);
}
```

**Abbildung 4.1:** Das Programm „rectangle“ erzeugt unter RT-Linux ein Rechtecksignal am Parallelport

# 5 Das kommerzielle Echtzeit-Betriebssystem LynxOS

## 5.1 Eigenschaften von LynxOS

LynxOS ist ein UNIX-Derivat, das speziell für Realzeit-Anwendungen entwickelt wurde. Es unterstützt die UNIX-Standards

- BSD 4.4
  - UNIX IPCs
  - Netzwerk API (sockets)
- POSIX.1a OS Definition (siehe Abschnitt 7)
- POSIX.1c Threads Extensions (siehe Abschnitt 8)
- POSIX.1b Real-Time Extensions (siehe Abschnitt 9)

Außerdem unterstützt es folgende Prozessoren:

- Intel x86
- Motorola/IBM PowerPC
- Motorola 68K
- MIPS R5000 Familie
- SME microSPARC I und II

Natürlich unterstützt es auch eine Reihe von System-Komponenten, die von einem normalen UNIX angeboten werden. Die folgende System-Komponente soll verdeutlichen, dass LynxOS ein vollständiges UNIX darstellt.

- (Fast-)Ethernet, ATM, ISDN, $I^2C$, SCI (Scalable Coherent Interface)
- PCI, CompactPCI, PC/104, VME/Eurobus
- TCP/IP, DNS, NFS
- X/Motif, Java
- Streams

Es stellt sich die Frage, ob ein solch komplettes Betriebssystem überhaupt für Realzeit-Anwendungen geeignet ist, da die meisten Realzeit-Anwendungen in eingebetteten Systemen benötigt werden, die wenige Ressourcen besitzen. LynxOS ist aber modular aufgebaut. Es kann so umkonfiguriert werden, dass nur die unbedingt benötigten Komponenten vom System unterstützt werden. Linux ist ebenfalls modular aufgebaut. Dort kann man die Quellen des Kerns konfigurieren und neu compilieren. LynxOS liefert

die Quellen natürlich nicht mit. Wie bei Windows CE stehen dem Entwickler aber alle Objekt-Dateien zu Verfügung. Durch ein Konfigurations-Tool werden die benötigten Objekt-Dateien ausgewählt und zu einem neuen Kern gelinkt. Die kleinste Konfiguration des Kerns benötigt weniger als 33 KByte Arbeitsspeicher. Der Hersteller von LynxOS beschränkt sich hauptsächlich bei der Entwicklung des Betriebssystems auf die Pflege des Kerns. Die meisten Programme, die dem Betriebssystem beiliegen, stammen aus der OpenSource-Bewegung. Sie wurden nur teilweise erweitert, um sie an die speziellen Erfordernisse eines Realzeit-systems anzupassen. Somit stehen die bekannten Programme wie GCC, GDB, GPROF, DDD und GNU Make zur Verfügung. Teilweise wurden einige System-Komponenten (Motif, Insure) hinzu gekauft. Aus diesem Grund kann sich die Firma auf die Entwicklung der Realzeit-Eigenschaften des Systems konzentrieren, und LynxOS kann trotzdem mit einer ausgereiften Entwicklungsumgebung aufwarten.

Bei Realzeit- und eingebetteten Systemen wird oft selbst entwickelte Hardware verwendet. Aus diesem Grund ist es für ein Realzeitsystem wichtig, dass die Entwicklung von Treibern relativ einfach vonstatten geht. Daher werden alle zur Treiber-Entwicklung benötigten Header und Libraries mitgeliefert (was leider keine Selbstverständlichkeit ist). Außerdem liefert Lynx alle Treiber, die sie selber entwickelt haben, mit Sourcen aus. Dort kann sich jeder anschauen, wie zum Beispiel die Treiber für eine Netzwerkkarte implementiert wurden und daraus Schlüsse für die eigene Entwicklung ziehen. Im Gegensatz zu den meisten anderen Betriebssystemen existieren unter LynxOS sogar Threads auf Kern-Ebene. Zum Beispiel können Threads bei der Entwicklung von Treibern verwendet werden. Durch die Verwendung von Kernel-Threads kann das Problem der Prioritäten-Inversion vermieden werden. Mehrere Programme können gleichzeitig mittels eines Systemcalls in den Kern eintreten. Dort übernehmen Kernel-Threads nicht nur die Parameter, die beim Aufruf des Systemcalls übergeben werden, sondern auch die Priorität des aufrufenden Programms. Ein solcher Mechanismus existiert bei den meisten anderen Betriebssystemen nicht.

## 5.2 Die Speicherverwaltung von LynxOS

LynxOS bietet eine virtuelle Speicherverwaltung mit Demand Paging an. Ziel eines Realzeitsystems ist es, eine zeitliche Vorhersagbarkeit der Ereignisse zu erreichen. Dies kann durch Demand Paging nicht gewährleistet werden, da es kaum vorherzusagen ist, ob ein Speicherstück eines Programmes ausgelagert ist oder nicht. Aus diesem Grund kann das Demand Paging durch Umkonfiguration des Kerns abgeschaltet werden, wodurch der Kern noch kleiner wird.

Wie die meisten Betriebssysteme, bietet LynxOS eine virtuelle Speicherverwaltung an, d.h. jeder Prozess besitzt seinen eigenen virtuellen Adressraum. Dadurch kann kein Prozess in den Adressraum eines anderen Prozesses schreiben. (Eine Ausnahme stellen die gemeinsamen Speicherstücke aus dem Abschnitt 7.4 dar.) Diese Eigenschaft des virtuellen Adressraumes fördert die Stabilität des Systems, welche gerade in Realzeitsystemen benötigt wird. Bei den meisten Betriebssystemen besitzt der Kern auch einen eigenen virtuellen Adressraum. Dies stellt aber bei einigen Systemcalls ein Problem dar. Angenommen, man möchte mit Hilfe von Sockets ein Datenpaket über das Netzwerk verschicken, so wird der folgende Befehl verwendet:

```
ssize_t send(int s, const void *msg, size_t len, int flags);
```

Dem Betriebssystem wird neben der Socket-Id *s*, der Paketlänge *len* und einigen Flags auch die virtuelle Adresse *msg*, an der das zu übertragende Paket steht, übermittelt. Das Betriebssystem muss nun auf die Daten zugreifen. Leider stammt die Adresse *msg* aus einem anderen virtuellen Adressraum und der Kern kann daher mit Hilfe dieser Adresse nicht auf die Daten zugreifen. Der Kern kann aber die virtuelle Adresse in eine Adresse umwandeln, die er interpretieren kann, und anschließend auf die Daten zugreifen. Das Umwandeln der virtuellen Adresse kostet aber Zeit. Die meisten Betriebssysteme verwenden jedoch diese Zugriffsart, da es die Systemsicherheit fördert. Bei Realzeitbetriebssystemen sollen die Kosten eines Systemaufrufs äußert gering gehalten werden. Aus diesem Grund wurde das Speichermodell von LynxOS leicht abgeändert. Abbildung 5.1 soll die Speicherverwaltung von LynxOS verdeutlichen. Im oberen Teil des 4 GByte

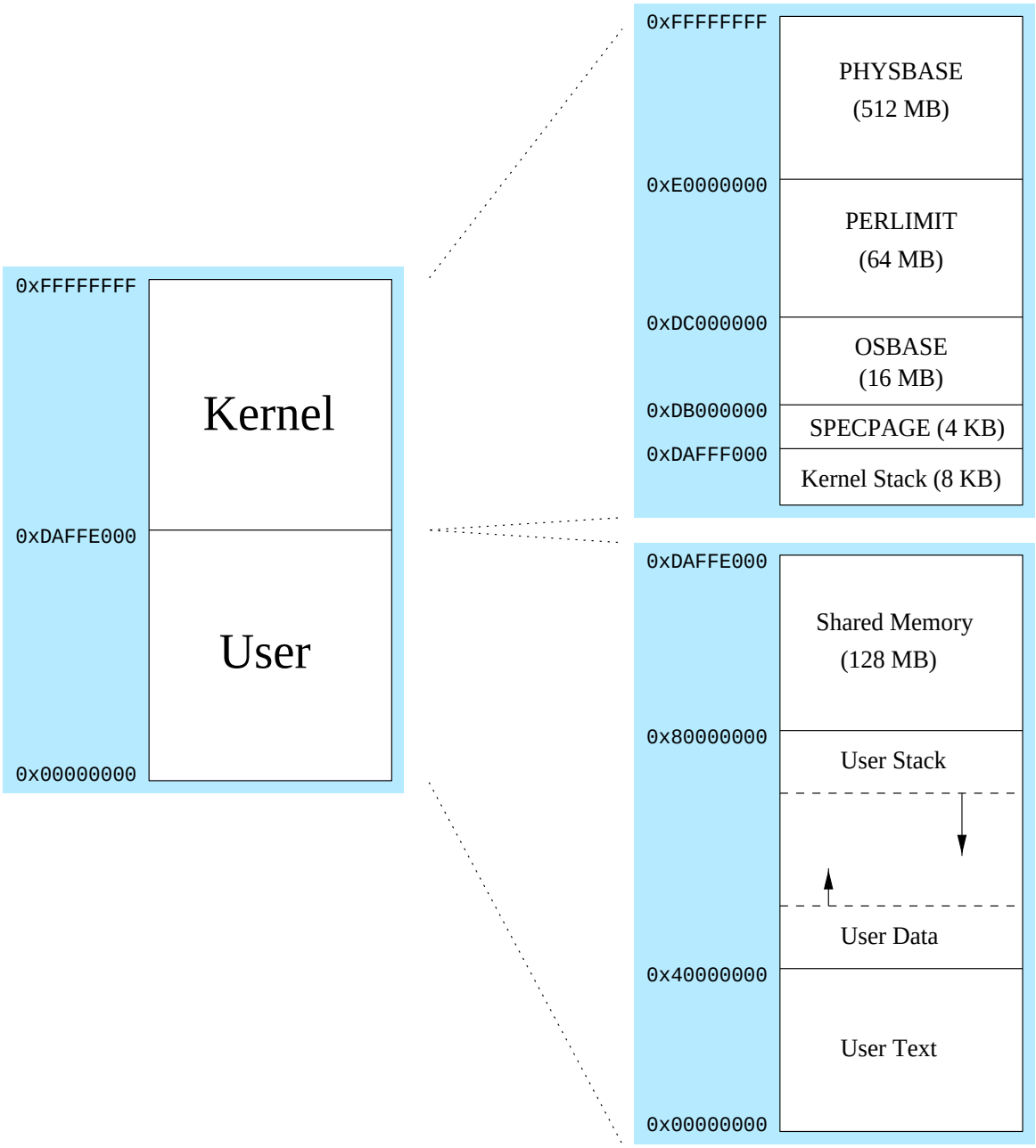



**Abbildung 5.1:** Virtueller Adressraum eines Programmes unter LynxOS

großen virtuellen Adressraums eines Programmes blendet sich der Kern ein. Da sich der Kern im virtuellen Adressraum des Programmes befindet, kann er, ohne die virtuelle Adresse in eine physikalische Adresse umzurechnen, beliebig im Adressraum des Programmes zugreifen. Dadurch wird Zeit eingespart und die Systemcalls können besser verarbeitet werden. Theoretisch könnte ein Programm auf den Kern zugreifen, da er sich in seinem virtuellen Adressraum befindet. Doch einzelne Seiten können durch die MMU (Memory Managemnet Unit) vor dem Zugriff des Programmes geschützt werden. Würde also ein Programm lesend oder schreibend auf eine Speicherstelle des Kerns zugreifen, so würde dieses einen Page-Fault auslösen. Somit kann ein unkontrollierter Zugriff eines Programmes auf den Speicher des Kerns vermieden werden. Die Adressräume der verschiedenen Anwendungsprogramme sind weiterhin vollkommen voneinander getrennt.

## 5.3 Threads auf der Ebene des Kerns

Angenommen, mehrere Tasks möchten Daten von einer Festplatte einlesen und das Gerät ist nicht bereit, Befehle entgegen zu nehmen, so werden die Anfragen der Tasks in einer Queue abgelegt und die Tasks blockieren, bis die Anfrage bearbeitet ist. Das Gerät signalisiert durch einen Interrupt, dass es bereit ist, Anfragen entgegen zu nehmen. Nun wird ein normales Betriebssystem in seiner Queue nachschauen, ob dort weitere Anfragen aufgelistet sind. Ist dies der Fall, wird eine neue Anfrage gestartet. Angenommen, es existieren Prozesse mit niedriger Priorität, die ständig Anfragen erzeugen, so lösen diese Prozesse durch ihre Anfragen ständig Interrupts aus. Existiert im System ein Prozess mit hoher Priorität, welcher aber nicht auf das Gerät zugreift, so würde dieser Prozess durch die Interrupts unterbrochen, obwohl die Interrupts für einen Prozess bestimmt sind, welcher eine wesentlich geringere Priorität besitzt. Sind die Anfragen einmal in der Queue eingetragen, werden sie auf jeden Fall abgearbeitet, obwohl ein Prozess mit höherer Priorität existiert, da das Betriebssystem die Queue so schnell wie möglich leeren will. Somit geht die Priorität einer Anfrage im Kern verloren. Löst ein Gerät einen Interrupt aus, so muss dieser Interrupt umgehend bearbeitet werden. Leider kann man an einem Interrupt nicht erkennen, ob dieser nicht so wichtig ist und erst dann bearbeitet wird, wenn der wichtigere Prozess beendet bzw. blockiert ist.

Wie bereits erwähnt, existieren im Kern von LynxOS sogar Threads. Diese Threads werden als Kernel-Threads bzw. System-Threads bezeichnet und sind nicht mit den Threads aus Abschnitt 8 zu verwechseln, da System-Threads nur im Kern arbeiten. Das obige Problem versucht man in LynxOS mit Hilfe dieser Threads zu lösen. Angenommen, mehrere Tasks möchten Daten von einer Festplatte einlesen und das Gerät ist nicht bereit, Befehle entgegen zu nehmen, so werden die Anfragen der Tasks in einer Queue abgelegt. Diese Queue wird von einem Kernel-Thread abgearbeitet. Trägt ein Task eine Anfrage in diese Queue ein, so schaut er nach, ob die Priorität des Kernel-Threads kleiner als seine eigene ist. Ist dies der Fall, erhöht der Task die Priorität des Kernel-Threads. Ist das Gerät bereit, eine Anfrage entgegen zu nehmen, löst es einen Interrupt aus. Der zugehörige Interrupt-Handler weckt den Thread auf, der die Queue verwaltet. Dieser Kernel-Thread nimmt eine Anfrage aus der Queue, leitet sie an das entsprechende Gerät weiter und legt sich anschließend wieder schlafen. Existiert ein Prozess mit sehr hoher Priorität, der gleichzeitig rechenbereit ist wie der Kernel-Thread, so kann der Scheduler anhand der Priorität entscheiden, dass der Kernel-Thread, der die Queue verwaltet, nicht so wichtig wie der Prozess mit sehr hoher Priorität ist. Somit weist der Scheduler dem wichtigeren Prozess Rechenzeit zu. Durch dieses Verfahren

wird vermieden, dass unwichtige Anfragen an ein Gerät gesendet werden, die sonst Interrupts auslösen könnten, die die wichtigeren Prozesse stören. Abbildung 5.2 veran-

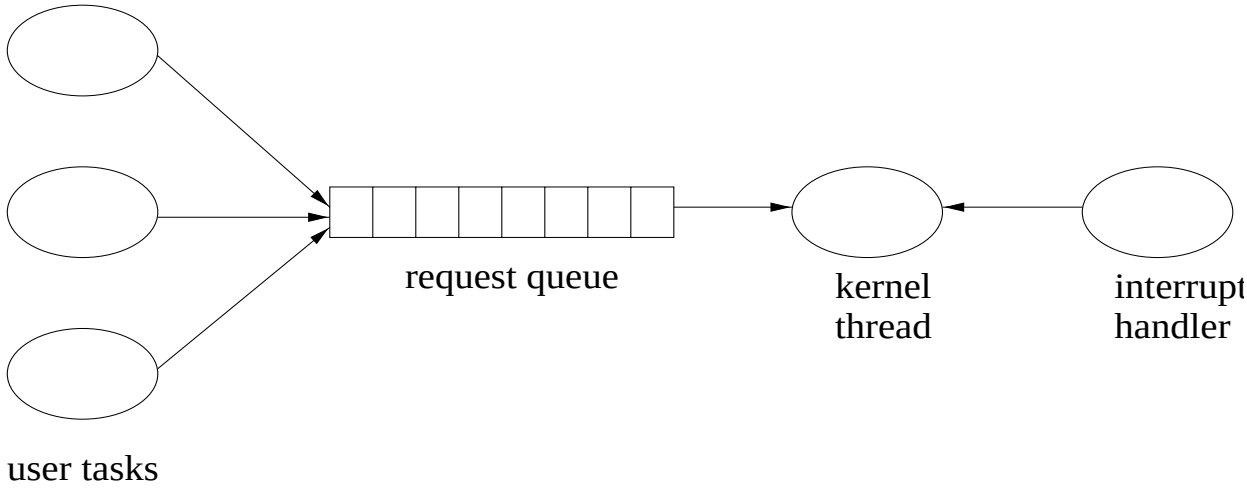



**Abbildung 5.2:** Beispiel für die Verwendung von Kernel-Threads

schaulicht nochmals die Vorgehensweise von LynxOS.

Im Abschnitt 9.4 wurde erläutert, wie man die Schedulingeigenschaften ändern kann. Intern werden die verschiedenen Prioritäten und Scheduling-Klassen in LynxOS auf Prioritätsstufen zwischen 0 und 255 abgebildet. Der Kern kennt aber insgesamt 512 Prioritätsstufen. In Abbildung 5.3 wird erläutert, wie die unterschiedlichsten Prioritätsstufen

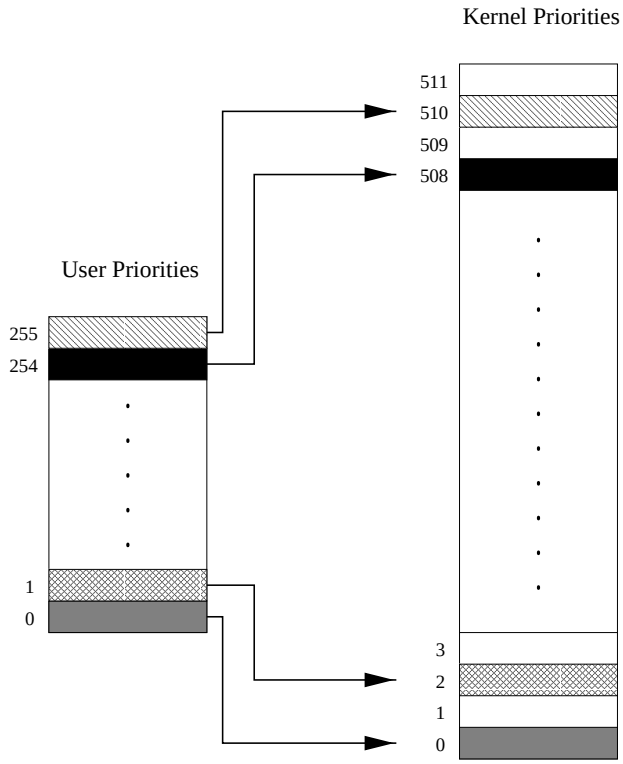



**Abbildung 5.3:** Abbildung der Anwender-Prioritäten auf die Kern-Prioritäten

auf der Ebene des Anwenders auf der Ebene des Kerns abgebildet werden. Die Prioritäten auf Anwenderebene werden mit zwei multipliziert, um die Prioritäten auf der Ebene

des Kerns zu erhalten. So entsteht zwischen den benachbarten Prioritäten auf der Ebene des Anwenders eine Lücke auf der Ebene des Kerns. In diese Lücken können die Kernel-Threads hineinstoßen. Angenommen, ein Task möchte etwas von der Festplatte, die noch nicht zur Verfügung steht, auslesen, so schreibt er seine Anforderung in eine Queue, die einen Kernel-Thread verwaltet. Besitzt der Prozess eine höhere Priorität auf der Ebene des Kerns als der Kernel-Thread, so erbt dieser seine um Eins erhöhte Priorität. Dies hat den Vorteil, dass der Kernel-Thread von einem Prozess verdrängt werden kann, der eine höhere Priorität besitzt, als der Prozess, der die Anforderung gestellt hat, aber nicht von einem Prozess, der die gleiche Priorität besitzt wie der Prozess, der die Anforderung gestellt hat. Es ist für das Betriebssystem wichtiger, die Queue zu leeren, als Prozessen Rechenzeit zu erteilen, die keine höhere Priorität besitzen.

## 5.4 Die Entwicklungsumgebung von LynxOS

Einige Versuche, die in diesem Praktikum durchgeführt werden, verwenden das Realzeitbetriebssystem LynxOS. Leider stehen dem Lehrstuhl nicht genügend Rechner zur Verfügung, damit jeder Teilnehmer an seinem eigenen LynxOS-Rechner arbeiten kann. Im Praktikumsraum befinden sich nur vier PCs, die mit LynxOS ausgestattet sind. Aus diesem Grund stellt der Lehrstuhl für Betriebssysteme neben dem gcc auf den Versuchsrechnern einen Cross-Compiler für LynxOS zur Verfügung. Jeder Teilnehmer sollte seine Programme auf einer Workstation von Sun editieren und compilieren. Zum Editieren wird am besten *XEmacs* verwendet, aber auch andere Text-Editoren können verwendet werden. Bei dem Cross-Compiler handelt es sich um einen *gcc*, den man unter */usr/lynx/3.0.1/x86/cdk/sunos-coff-x86/bin* findet. Man sollte beim Compilieren darauf achten, dass man den Cross-Compiler verwendet und nicht den *gcc*, der Code für Sun-Workstation erzeugt. Das Austesten der Programme kann auf dem eigentlichen LynxOS-Rechner durchgeführt werden. Diese Vorgehensweise entlastet die LynxOS-Rechner des Lehrstuhls. Für zeitkritische Aufgaben sollte jeder seine Lösung ausgiebig testen. Ist man der Meinung, dass die Lösung korrekt funktioniert, sollte durch die eigene Absprache mit den Teilnehmern dafür gesorgt werden, dass kein anderer Teilnehmer während des Testlaufs den Rechner benutzt, da sonst die Messungen verfälscht werden.

Stürzt das entwickelte Programm auf einem LynxOS-Rechner ab, kann man es dort mit dem *gdb* oder *ddd* debuggen. Ausführliche Beschreibungen dieser Programme sind im Kapitel F und G zu finden.

# 6 Das kommerzielle Echtzeit-Betriebssystem QNX

## 6.1 Eigenschaften von QNX

Auf den Kern eines Betriebssystems wird üblicherweise Bezug genommen, ohne weiter zu hinterfragen, um welchen Anteil am Betriebssystem es sich dabei handelt. Einigkeit besteht darüber, dass der Kern für ein Betriebssystem zentrale Bedeutung hat und der zugehörige Code sehr häufig aufgerufen und ausgeführt wird. Mindestens für die folgenden Aufgaben ist der Kern zuständig:

- die Verwaltung der Datenstrukturen zur Beschreibung eines Prozesses
- die Verarbeitung von Systemaufrufen
- die Zuteilung des Prozessors an einen rechnenden Prozess

Zur Bewältigung dieser Aufgabenstellungen greift der Kern auf eine Reihe zustandsbeschreibender Daten zu. Da nur konsistente Zustandsänderungen zulässig sind, muss der Zugriff auf zustandsbeschreibende Datenstrukturen geschützt werden, d.h. eine Unterbrechung beim Zugriff auf zustandsbeschreibende Datenstrukturen wird verhindert.

Kerne, die keine Unterbrechungen zulassen, können die Reaktion auf externe Ereignisse oder auf Systemaufrufe unvorhersehbar verzögern und somit die Einhaltung harter Zeitbedingungen unmöglich machen. Deshalb zielt eine zentrale Forderung dahin, den Kern von Realzeitbetriebssystemen unterbrechbar zu machen.

Ein Betriebssystem zu entwickeln, bei dem der Kern (fast) zu jeder Zeit unterbrechbar ist und gleichzeitig alle zustandsbeschreibenden Datenstrukturen schützt, ist äußerst schwierig. Besonders schwierig ist die Aufgabe bei monolytischen Betriebssystemen (z.B. LynxOS, Linux), da der Umfang eines solchen Systems ein großes Problem darstellt.

Um die Ausgabe zu vereinfachen, wird bei der Entwicklung von Realzeitbetriebssystemen eine Mikrokern-Architektur (siehe Abbildung 6.1) verwendet. Ein typischer Vertreter dieser Gattung von Realzeitbetriebssystemen stellt *QNX* dar. Bei einer Mikrokern-Architektur werden nur die wichtigsten Komponenten im Kern realisiert. Zum Beispiel werden Treiber oder das Dateisystem nicht im Kern implementiert, sondern laufen als eigenständiger Task im System. Dies hat zur Folge, dass die Interprozess-Kommunikation (IPC) eine äußerst wichtige Komponente darstellt. Aus diesem Grund wurden in QNX einige neuartige Interprozess-Kommuniktionsmöglichkeiten entwickelt. In diesem Praktikum wird aus Zeitgründen nicht auf die QNX-spezifischen IPCs eingegangen. In der Vorlesung *„Realzeitsysteme I“* werden diese Möglichkeiten aber ausführlich erläutert. Einige Informationen findet man aber auch im Internet unter *http://www.qnx.com*

QNX ist speziell für PC-Architekturen entwickelt worden und unterstützt alle gängigen Hardware-Komponenten. Es ist ein UNIX-Derivat, dass kompatibel zu POSIX.1b ist. Leider definiert dieser Standard nur die fundamentalsten Komponenten eines UNIX-

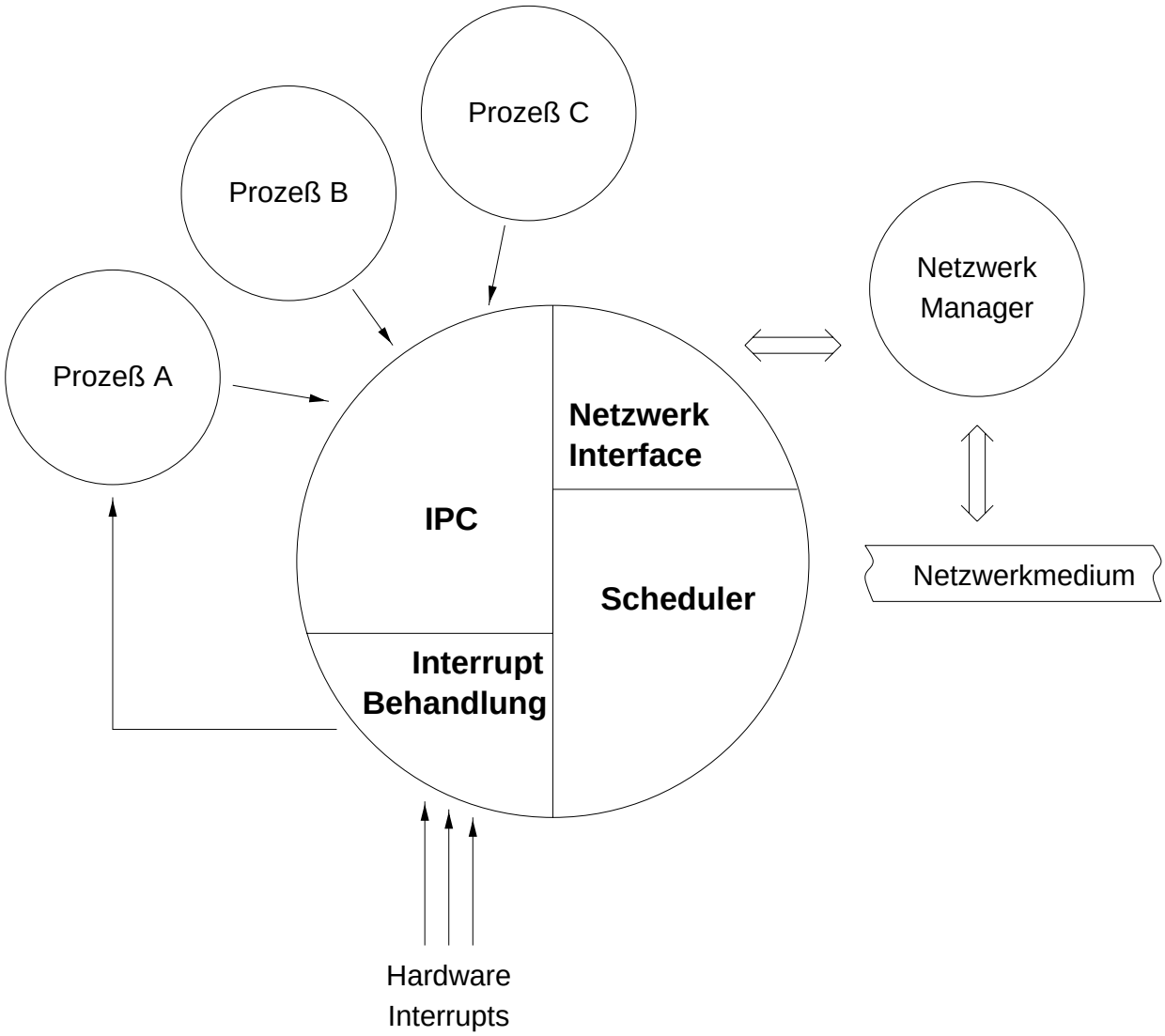



**Abbildung 6.1:** QNX-Mikrokern

Betriebssystems. QNX existiert in zwei Versionen: QNX4 und QNX 6 / RTP (Neutrino). QNX 4 ist nur teilweise kompatibel mit POSIX.4 und kennt keine Threads. QNX 6 ist voll-kompatibel mit POSIX.4, und unterstützt Threads. Ausserdem ist es moderner und umsonst. QNX generell ist wegen seiner Architektur relativ gewöhnungsbedürftig, bietet aber eine Reihe neuartiger Befehle, die in der Realzeitverarbeitung gewünscht werden. Es folgt ein Beispiel für die Timerprogrammierung unter QNX

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>

timer_t             tid;
```

```
void ExitHandler(int sig)
{
    /* delete timer */
    timer_delete(tid);
    /* exit process */
    exit(0);
}

/*************************************************************/
/* handler function after timer expiration to update the time */
/*************************************************************/
void ALRM_Handler(int sig)
{
    static int i = 0;
    i++;
    fprintf(stderr, "%i. Iterationsschritt: Hello world !!!   sig =
    %i\n", i, sig);
}

int main(int argc, char** argv)
{
    struct itimerspec    new_setting, old_setting;
    sigset_t             signals;
    struct sigaction     sINT, sTERM, sALRM;
    int                  i;

    /* set signal mask for sigsuspend */
    sigemptyset(&signals);

    /* define action for SIGINT */
    sINT.sa_handler = ExitHandler;
    sINT.sa_flags = 0;
    if( sigaction(SIGINT, &sINT, NULL) < 0 )
    {
        fprintf(stderr, "Error: sigaction\n");
        exit(1);
    }

    /* define action for SIGTERM */
    sTERM.sa_handler = ExitHandler;
    sTERM.sa_flags = 0;
    if (sigaction(SIGTERM, &sTERM, NULL) < 0)
    {
        fprintf(stderr, "Error: sigaction\n");
        exit(1);
```

```
    }

    /* define action for SIGALRM */
    sALRM.sa_handler = ALRM_Handler;
    sALRM.sa_flags = 0;
    if( sigaction(SIGALRM, &sALRM, NULL) < 0 )
    {
        fprintf(stderr, "Error: sigaction\n");
        exit(1);
    }

    /* create timer */
    tid = timer_create(CLOCK_REALTIME, NULL);
    if (tid < 0)
    {
        perror("timer_create");
        exit(1);
    }

    /* set timer values */
    new_setting.it_value.tv_sec = 1;
    new_setting.it_value.tv_nsec = 0;
    new_setting.it_interval.tv_sec = 0;
    new_setting.it_interval.tv_nsec = 500000000;

    /* start timer */
    i = timer_settime(tid, 0, &new_setting, &old_setting);
    if (i < 0)
    {
        perror("timer_settime");
        exit(1);
    }
    for(i=0; i<100; i++)
    {
        /* wait for timer expiration */
        sigsuspend(&signals);
    }
    /* delete timer */
    timer_delete(tid);
    return 0;
}
```

**Abbildung 6.2:** Beispiel für einen periodischen Tmer unter QNX 4

In Kapitel 10 wird ein Verfahren vorgestellt, wie die Interrupt-Latenzzeiten auf einfache

Weise ermittelt werden können. In diesem Zusammenhang werden ein paar QNX-spezifische Befehle vorgestellt, die dieses sehr leicht realisieren. Man wird sehen, dass eine hardwarenahe Programmierung unter QNX äußerst einfach ist.

QNX 6 ist POSIX.4 und POSIX 1.c kompatibel und ist daher erheblich einfacher zu programmieren. Daher wird im Praktikum nur QNX 6 verwendet, auf dem die POSIX Beispielprogramme laufen. Für QNX 6 ist desweiteren ein gcc erhältlich, was Portierungsaspekte des Codes weiter vereinfacht. Der in diesem Kapitel vorgestellte Quellcode ist daher für das Praktikum nicht mehr direkt von Interesse.

# 7 Systemprogrammierung unter UNIX

Für einige Realzeit-Versuche ist eine systemnahe Programmierung notwendig. Da in diesem Praktikum hauptsächlich UNIX-basierte Systeme eingesetzt werden, welche der Single UNIX Specification folgen, wird nun eine kleine Einführung in die Systemprogrammierung unter UNIX gegeben. Dies stellt nur einen kleinen Einblick in die Systemprogrammierung unter UNIX dar. Weiterführende Informationen findet man unter [5] oder bei *The Open Group*.

## 7.1 Signals unter UNIX

Eine der ältesten Möglichkeiten der Interprozesskommunikation unter UNIX sind Signale (engl. signals). Der Kern benutzt Signale, um Prozesse über bestimmte Ereignisse zu informieren. Der Anwender benutzt Signale in der Regel, um Prozesse abzubrechen oder interaktive Programme in einen definierten Zustand zu überführen. Der Anwender kann zum Beispiel das Programm abbrechen, indem er die Tastenkombination CTRL-C drückt. Dadurch sendet er implizit dem Programm das Signal *SIGINT*, worauf das Programm abbricht. Ein weiteres sehr bekanntes Signal ist *SIGTERM*. Der Anwender sendet dieses Signal zum Programm, indem er in einer Konsole *kill pid* eingibt, wobei *pid* für die Prozess-Id des Programmes steht. Das Programm wird durch diesen Befehl aufgefordert zu terminieren.

Der Programmierer kann jedes Signal, das sein Programm erhält, abfangen und einen Signal-Handler definieren, der dieses Signal verarbeitet. Im folgenden Beispielprogramm sollen die Signale *SIGINT* (also die Tastenkombination CTRL-C) und *SIGTERM* abgefangen werden, eine Ausgabe auf den Bildschirm schreiben und anschließend terminieren.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

/******************/
/* signal handler */
/******************/
void Handler(int sig)
{
  fprintf(stderr, “Hello world !!! sig = %i\n“, sig);
  exit(0);
}

int main(int argc, char** argv)
{
    struct sigaction     sINT, sTERM;
```

```
    /* define action for SIGINT */
    sINT.sa_handler = Handler;
    sINT.sa_flags = 0;
    if( sigaction( SIGINT, &sINT, NULL) < 0 )

    {
        fprintf(stderr, “Error: sigaction\n“);
        exit(1);
    }

    /* define action for SIGTERM */
    sTERM.sa_handler = Handler;
    sTERM.sa_flags = 0;
    if( sigaction( SIGTERM, &sTERM, NULL) < 0 )
    {
        fprintf(stderr, “Error: sigaction\n“);
        exit(1);
    }

    for(;;) /* infinte loop */
        ;
}
```

**Abbildung 7.1:** Beispielprogramm zum Abfangen von Signals

Zunächst muss man definieren, welche Funktion aufgerufen werden soll, wenn das Signal *SIGINT* bzw. *SIGTERM* ausgelöst wird. In der Datenstruktur *struct sigaction* initialisiert man hierzu die Variable *sa_handler* auf die Adresse der aufzurufenden Funktion. In diesem Beispiel wurde die Adresse der Funktion *Handler* angegeben. Durch die Funktion *sigaction* teilt man nun dem Betriebssystem mit, das es bei dem Signal *SIGINT* bzw. *SIGTERM* den Signal-Handler *Handler* aufrufen soll. Hierzu wird der Funktion als Argumente *SIGINT* bzw. *SIGTERM* und die Datenstruktur *struct sigaction* übergeben, die zuvor initialisiert wurde. Damit das Programm nicht nach dem Erzeugen des Signal-Handlers terminiert, wurde zum Schluss eine Endlos-Schleife implementiert. Das Programm terminiert nur, wenn die Funktion *exit* aus dem Signal-Handler aufgerufen wird.

Jedem Signal ist eine eindeutige Nummer zugeordnet. Wird ein Signal ausgelöst, so wird der entsprechende Signal-Handler aufgerufen. Im Beispielprogramm wird aber nur ein Signal-Handler für *SIGINT* bzw. *SIGTERM* definiert. Damit der Signal-Handler unterscheiden kann, welches der beiden Signale gerade aufgetreten ist, wird ihm vom Betriebssystem auch die Nummer des Signals übergeben.

## 7.2 Periodische Timer unter UNIX

Jedes UNIX-Betriebssystem stellt einen Timer zur Verfügung, der periodisch das Signal *SIGALRM* zum Programm sendet. Ein Beispielprogramm für die Programmierung eines solchen periodischen Timers findet man in Abbildung 7.2. Zunächst wird wie in Abbildung 7.1 ein Signal-Handler für das Signal *SIGALRM* definiert. Anschließend muss man

nur noch den periodischen Timer starten, der das Signal *SIGALRM* periodisch auslöst. Dies geschieht durch die Funktion *setitimer* nach . Als erster Parameter muss hierfür die Konstante *ITIMER_REAL* übergeben werden. Als zweiter Parameter wird die Adresse einer Datenstruktur vom Typ *struct itimerval* übergeben. Diese Datenstruktur besteht aus zwei Variablen vom Typ *struct timeval*. Die erste mit dem Namen *it_value* beschreibt den Zeitpunkt, an dem der periodische Timer gestartet wird, während die zweite mit dem Namen *it_interval* die Periodenlänge beschreibt. Jede Variable besteht wieder aus zwei Variablen vom Typ *long*, wobei eine die Sekunden und die andere die Mikrosekunden beschreibt. Im Beispiel aus der Abbildung 7.2 wird der periodische Timer so initialisiert, dass er nach einer Sekunde gestartet und alle 0.5 Sekunden erneut ausgelöst wird.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#ifdef __linux__
#include <sys/time.h>
#else
#include <time.h>
#endif
#include <sys/types.h>
#include <sys/stat.h>

/*************************************************************/
/* handler function after timer expiration to update the time */
/*************************************************************/
void ALRM_Handler(int sig)
{
    static int i = 0;

    i++;
    fprintf(stderr, “%i. Iterationsschritt: Hello world !!!   sig,
            %i\n“, i, sig);
}

int main(int argc, char** argv)
{
  sigset_t             signals;       /* define the signal mask */
  struct itimerval     values,oldvalues;/* timer values */
  struct sigaction     sALRM;     /* define the action for signal
                                          SIGALRM */
  int i;

    /* set signal mask for sigsuspend */
    sigemptyset(&signals);

    /* define action for SIGALRM */
```

```
    sALRM.sa_handler = ALRM_Handler;
    sALRM.sa_flags = 0;
    if( sigaction( SIGALRM, &sALRM, NULL) < 0 )
    {
        fprintf(stderr, “Error: sigaction\n“);
        exit(1);

    }

    /* set timer values */
    values.it_value.tv_sec = 1;
    values.it_value.tv_usec = 0;
    values.it_interval.tv_sec = 0;
    values.it_interval.tv_usec = 500000;

    /* start timer */
    setitimer( ITIMER_REAL, &values, &oldvalues );

    for(i=0; i<100; i++)
    {
          /* wait for timer expiration */
          sigsuspend(&signals);
    }

    exit(0);
}
```

**Abbildung 7.2:** Beispiel für die Verwendung eines Timers

In diesem Beispielprogramm wurde kein Signal-Handler für die Signale *SIGTERM* und *SIGINT* definiert. Aus diesem Grund wird der Standard-Signal-Handler benutzt, wenn *SIGTERM* oder *SIGINT* ausgelöst werden. Dieser Standard-Signal-Handler lässt das Programm terminieren.

Das Beispielprogramm soll genau einhundertmal auf das Signal warten. Dies wurde durch eine Schleife implementiert, die einhundertmal durchlaufen wird. In dieser Schleife suspendiert das Programm durch den Befehl *sigsuspend* solange, bis ein Signal ausgelöst wird. Durch eine Variable vom Typ *sigset_t* wird dem Befehl *sigsupend* mitgeteilt, auf welche Signale das Programm nicht reagieren soll. Da das Programm aber auf alle Signale reagieren soll, wurde diese Variable durch die Funktion *sigemptyset* initialisiert. Signale, auf die das Programm keine Reaktion zeigen soll, werden durch die Funktion *sigaddset* hinzugefügt.

## 7.3 Interprozesskommunikation durch sogenannte „Named Pipes“

Neben Signals können Prozesse auch über *„named pipes“* miteinander kommunizieren. Man kann sich eine *„named pipe“* als eine Röhre vorstellen. Auf jeder Seite der Röhre lauscht ein Prozess. Jeder Prozess kann eine Nachricht in die Röhre legen, so dass sie

auf der anderen Seite ankommt. Dabei kommt stets die zuerst gesendete Nachricht auf der anderen Seite der Röhre an. Eine solche Reihenfolge nennt man FIFO (First In First Out). Unter UNIX werden solche Pipes als eine spezielle Form von Dateien behandelt, d.h. jede Pipe erscheint im Dateisystem und kann wie eine Datei gelesen und gelöscht werden. Bevor man eine Pipe erzeugt, sollte gewährleistet sein, dass alle alten Pipes, die den gleichen Pfad verwenden, gelöscht sind. Dies kann durch folgenden Befehl realisiert werden:

```
int unlink(const char* path);
```

Eine Pipe wird durch den folgenden Befehl erzeugt:

```
int mkfifo(const char *path, mode_t mode);
```

Der Name der Pipe bzw. Pfad der Datei wird dem Befehl durch die Variable *path* übergeben. Für die Variable *mode* sollte die Konstante *S_IRWXU* (definiert im Header *sys/stat.h*) eingetragen werden. Dadurch bekommt der Anwender auf die Datei bzw. Pipe Read-, Write- und Execute-Rechte. Nach dem Erzeugen der Pipe kann ein Programm die Pipe öffnen, um aus ihr Daten zu lesen. Die Pipe wird wie eine Datei durch den folgenden Befehl geöffnet:

```
int open(const char *path, int oflag);
```

Den Datei- bzw. Pipenamen übergibt man hier ebenfalls durch die Variable *path*. Zusätzlich wird dem Betriebssystem durch den Parameter *oflag* mitgeteilt, ob aus der Pipe nur gelesen *(O_RDONLY)*, nur geschrieben *(O_WRONLY)* oder gelesen und geschrieben *(O_RDWR)* wird. Als Rückgabewert bekommt man einen Filedeskriptor, mit dem aus der Datei bzw. Pipe durch den folgenden Befehl gelesen wird:

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

In der Variablen *fildes* wird der Filedeskriptor der Pipe eingetragen, aus dem man lesen möchte. Durch *buf* wird der Funktion die Adresse übergeben, ab der die empfangenden Daten eingetragen werden. Die Größe des Puffers wird durch die Variable *nbyte* angegeben. Entsprechend zum Lesebefehl *read* existiert ein Schreibbefehl *write*, der die gleiche Syntax besitzt. Mit dem Befehl *close(filedes)* wird die Pipe mit dem Filedeskriptor *filedes* geschlossen.

## 7.4 Interprozesskommunikation über gemeinsamen Speicher

Die einfachste Möglichkeit zur Interprozesskommunikation stellt eigentlich der Speicher dar. Aus Sicherheitsgründen bieten aber alle modernen Betriebssysteme jedem Prozess einen eigenen virtuellen Adressraum an. Somit kann kein Prozess auf den Arbeitsspeicher eines anderen Prozesses zugreifen. Manchmal ist es aber vorteilhaft, dass einige Speicherstücke von mehreren Prozessen gemeinsam genutzt werden. Aus diesem Grund stellen die meisten Betriebssysteme einen Mechanismus zur Verfügung, über den gemeinsame Speicherstücke, sogenannte *„shared segments“*, angefordert werden.

Im folgenden Beispielprogramm sollen sich zwei Prozesse eine Integerzahl teilen, die jeder fünfzig Mal hochzählt. Da die Zahl auf Null initialisiert wurde, müsste anschließend die Zahl den Wert einhundert beinhalten.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#ifndef __Lynx__
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/fcntl.h>
#else
#include <ipc.h>
#include <shm.h>
#include <fcntl.h>
#endif
#include <sys/types.h>
#include <sys/wait.h>

const int SIZE = 4096;                /* size of shared segment */
const int KEY = 0xFEDCBA;

int main(int argc, char* argv[])
{
    int sid, pid, i = 0;
    volatile   int* counter;

    /* create shared segment */
    sid = shmget(KEY, SIZE*sizeof(int), O_RDWR | IPC_CREAT | 511);
    if (sid == -1)
    {
        perror("shmget");
        exit(1);
    }

    /* map segement into the virtual address space */
    if ((int) (counter = (int*) shmat(sid, 0, O_RDWR)) == -1)
    {
        perror("shmat");
        exit(1);
    }

    /* initialize counter */
    *counter = 0;

    /* create new child process */
    pid = fork();

    while(i < 50)
```

```
    {
        switch(pid) {
        case 0:  /* child process */
                 if ((*counter % 2 == 0) && (*counter < 100))
                 {
                     fprintf(stderr, "pid = %ii = %i*counter =
                     %i\n", (int) getpid(), i+1, *counter + 1);
                     i++;
                     (*counter)++;
                 }
                 break;

        default: /* parent process */
                 if ((*counter % 2 != 0) && (*counter < 100))
                 {
                     fprintf(stderr, "pid = %ii = %i*counter =
                     %i\n", (int) getpid(), i+1, *counter + 1);
                     i++;
                     (*counter)++;
                 }
                 break;
        }
    }

    /* parent wait for a child process to terminate */
    wait(NULL);

    if(pid != 0)
    {
        /* unmap shared segment */
        shmdt((void*) counter);

        /* remove shared segment  */
        shmctl(sid, IPC_RMID, 0);
    }

    return(0);
}
```

**Abbildung 7.3:** Beispielprogramm für die Benutzung von gemeinsamen Speicherstücken

Zunächst wird mit Befehl *shmget* ein Speichersegment erzeugt, das von mehreren Prozessen benutzt werden kann. Hierzu muss man dem Betriebssystem einen Schlüssel übergeben. Fordern zwei verschiedene Prozesse mit dem gleichen Schlüssel ein Speicherstück an, so erhalten beide Prozesse das gleiche Speichersegment, so dass sie über diesen Speicher miteinander kommunizieren können. Außerdem wird dem

Betriebssystem die Größe des Segments (hier 4096 Zahlen vom Typ *int*) und die Zugriffsart mitgeteilt. Die Zugriffsarten bei gemeinsamen Speichersegmenten entsprechen den Zugriffsarten bei Dateien, so dass man hier die gleichen Parameter wie bei dem Befehl *open* übergeben kann. In diesem Beispiel wird das gemeinsame Speichersegment so erzeugt, dass die Prozesse lesend und schreibend auf den Arbeitsspeicher zugreifen können. Außerdem schlägt der Befehl *shmget* fehl, falls bereits ein Speichersegment mit dem angegebenen Schlüssel existiert. Nachdem ein Speichersegment erzeugt wurde, muss es noch in den virtuellen Adressraum des Prozesses eingeblendet werden. Dies wird durch den Befehl *shmat* erreicht, indem dem Befehl die Segment-Nummer, die man vom Befehl *shmget* erhalten hat, einen Vorschlag für die virtuelle Adresse sowie die Zugriffsart übergeben wird. Im Beispielprogramm wurde für die virtuelle Adresse eine Null übergeben. Hierdurch kann sich das Betriebssystem eine beliebige Adresse für das Speichersegment auswählen. Der Rückgabewert der Funktion *shmat* beinhaltet die Adresse, an der das Betriebssystem das Segment tatsächlich eingeblendet hat.

Nun könnte ein zweiter Prozess gestartet werden, der mit dem gleichen Schlüssel ein Segment erzeugt und in seinem Adressraum einblendet. Es ist aber einfacher, einen Prozess vom laufenden Programm durch den Befehl *fork* abzuspalten. Dabei wird eine genaue Kopie vom laufenden Prozess erzeugt, d.h. alle Variablen und der Befehlszähler im neu erzeugten Prozess beinhalten den gleichen Wert wie der Original-Prozess. Somit setzen beide Prozesse nach dem Befehl *fork* das Programm fort. Dabei liefert *fork* beim neu erzeugten Prozess (auch Kind bzw. child genannt) Null zurück, während der andere Prozess (auch Vater bzw. parent genannt) die Prozess-Id des Child-Prozesses erhält. Nachdem die Prozesse die Aufgabe beendet haben, sollen sie terminieren. Hierbei sollte zunächst das eingeblendete Speichersegment durch *shmdt* ausgeblendet und anschließend wieder durch *shmctl* entfernt werden. Man muss aber darauf achten, dass beim Entfernen des Speichersegments nicht dem anderen Prozess der Arbeitsspeicher weggenommen wird, auf dem er gerade arbeitet. Dies kann am besten gewährleistet werden, indem der Parent-Prozess durch *wait* blockiert wird, bis der Child-Prozess terminiert. Anschließend kann der Parent-Prozess das Speichersegment ordnungsgemäß entfernen.

Beide Prozesse sollen einen gemeinsamen Zähler genau fünfzig Mal um eins erhöhen. Meistens möchte man die Programme optimieren. Dabei geht der Compiler davon aus, dass nur Prozess auf die Variablen zugreifen kann. Dies führt oft zu Problemen, wenn das Programm gemeinsame Speichersegmente benutzt, da hier mehrere Prozesse auf einer Variablen arbeiten können. Um das Problem zu umgehen, wird entweder die Optimierung abgeschaltet (was die Geschwindigkeit des Programms beeinträchtigt) oder dem Compiler dies durch das Schlüsselwort *volatile* mitgeteilt. Zum Beispiel bedeutet

```
volatile double *counter;
```

dass *counter* ein Zeiger ist, der auf einen Double-Wert zeigt und dieser Wert von einem anderen Prozess verändert werden kann. Ein weiteres Problem stellt die Synchronisation der Prozesse dar. Da beide den Zähler zunächst einlesen und anschließend erhöhen, kann dies zu falschen Ergebnissen führen. Angenommen, die Variable **counter* hat den Wert 10 und der Parent-Prozess würde zunächst den Wert einlesen und anschließend den Child-Prozess, dann würden beide den Wert 10 ins Register laden und ihn auf 11 erhöhen. Anschließend würden sie die Werte zurückschreiben und der Wert 11

würde im Speicher stehen. Doch der gewünschte Wert wäre 12 (= 10+1+1), da die beiden Prozesse den Wert um eins erhöhen. Um das Problem zu lösen, müsste man verhindern, dass zwischen dem Lesen und Schreiben des neuen Wertes ein Prozesswechsel durchgeführt wird. Im Beispiel aus Abbildung 7.3 ist das Problem gelöst, indem der Parent-Prozess alle ungeraden Werte und der Child-Prozess alle geraden Werte von **counter* erhöht. In dieser Lösung kann zwar ein Prozesswechsel zwischen dem Lesen und Schreiben eines neuen Wertes stattfinden, doch streiten sich die Prozesse nicht mehr um eine gemeinsame Ressource, da der eine Prozess nur die geraden und der andere nur die ungeraden Werte erhöht. Dies ist keine sonderlich elegante Lösung. Eine bessere Alternative wird im folgenden Abschnitt vorgestellt.

## 7.5 Standard UNIX-Semaphoren

Ein einfaches Hilfsmittel zur Synchronisation von Prozessen stellen Semaphoren dar. Semaphoren sind ganzzahlige Zahlen, die *atomar* hoch bzw. runter gezählt werden. Hierbei bedeutet atomar, dass zwischen dem Lesen der ganzzahligen Zahl und dem Schreiben kein Prozesswechsel stattfindet und in einen Schritt durchgeführt wird. Hierdurch wird das Problem aus Abschnitt 7.4 vermieden. Möchte man den Wert der Semaphore, die bereits den Wert Null hat, verringern, so blockiert der Prozess, bis ein anderer Prozess den Wert erhöht. D.h. der Wert der Semaphore kann niemals negativ sein.

Es existieren zwei Implementierungen von Semaphoren. Die ältere Implementierung stammt aus den frühen Tagen von UNIX. Die Verwendung dieser Semaphoren ist ein wenig ungewöhnlich, aber fast alle UNIX-Varianten besitzen diese Funktionen. Inzwischen existiert ein Standard für Realzeit-Systeme, der ebenfalls eine Semaphoren-Implementierung besitzt. Dieser Standard ist unter dem Namen POSIX.4 (offiziell POSIX 1.b) bekannt. Leider halten sich nicht alle Betriebssysteme an diesen Standard und unterstützen ihn gar nicht bzw. nur teilweise (Linux). Diese Semaphore und der dazu gehörige Standard wird in Abschnitt 9.3 genauer erläutert. Im folgenden Beispiel wird das Problem der Prozesssynchronisation aus Abbildung 7.3 durch die etwas veralteten Semaphoren gelöst.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#ifndef __Lynx__
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/fcntl.h>
#else
#include <ipc.h>
#include <shm.h>
#include <sem.h>
#include <fcntl.h>
#endif
#include <sys/types.h>
#include <sys/wait.h>
```

```
const int SIZE = 4*1024;        /* size of shared segment */
const int KEY = 0xFEDCBA;

#ifndef __Lynx__
union semun {
    int val;                    /* value for SETVAL */
    struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
    unsigned short int *array;  /* array for GETALL, SETALL */
    struct seminfo *__buf;      /* buffer for IPC_INFO */
};
#endif

int semaphore_wait(int sem_id)
{
    struct sembuf operation;

    operation.sem_num = 0;
    operation.sem_op = -1;
    operation.sem_flg = 0;

    return(semop(sem_id, &operation, 1));
}

int semaphore_post(int sem_id)
{
    struct sembuf operation;

    operation.sem_num = 0;
    operation.sem_op = 1;
    operation.sem_flg = 0;

    return(semop(sem_id, &operation, 1));
}

int main(int argc, char* argv[])
{
    int shmid, pid, semid, i = 0;
    volatile   int* counter;
    union semun arg;

    /* create shared segment */
    shmid = shmget(KEY, SIZE*sizeof(int),
                   O_RDWR | IPC_CREAT | 511);
    if (shmid == -1)
```

```
    {
        perror("shmget");
        exit(1);
    }

    /* map segement into the virtual address space */
    if ((int) (counter = (int*) shmat(shmid, 0, O_RDWR)) == -1)
    {
        perror("shmat");
        exit(1);
    }

    /* initialize counter */
    *counter = 0;

    /* create semaphore */
    semid = semget(KEY+1, 1, IPC_CREAT|O_RDWR|511);
    if (semid == -1)
    {
        perror("semget");

        exit(1);
    }

    /* initialize semaphore */
    arg.val = 1;
    if (semctl(semid, 0, SETVAL, arg) == -1)
    {
        perror("semctl");
        exit(1);
        }

    /* create new child process */
    pid = fork();

    while(i < 50)
    {
        semaphore_wait(semid);
        fprintf(stderr, "pid = %ii = %i*counter = %i\n",
        (int) getpid(), i+1, *counter + 1);
        (*counter)++;
        i++;
        semaphore_post(semid);
    }
```

```
    /* parent wait for a child process to terminate */
    wait(NULL);

    if (pid != 0)
    {
        /* remove semaphore */
        semctl(semid, 0, IPC_RMID, arg);

        /* unmap shared segment */
        shmdt((void*) counter);

        /* remove shared segment  */
        shmctl(shmid, IPC_RMID, 0);
    }

    return(0);
}
```

**Abbildung 7.4:** Prozesssynchronisation durch Standard-UNIX-Semaphoren

Das Anlegen der Semaphoren ähnelt dem Erzeugen von gemeinsamen Speichersegmenten. Durch den Befehl *semget* erzeugt man die Semaphore, wobei dem Befehl der Schlüssel, die Anzahl der Semaphoren und die Zugriffsart übergeben wird. Der Befehl *semget* gibt eine Semaphoren-Id zurück, über die man die einzelnen Semaphoren verwalten kann. Die Semaphoren werden mit Hilfe des Befehls *semctl* initialisiert. Hierzu wird eine Datenstruktur mit dem Namen *semun* benötigt. Der UNIX-Standard definiert, wie diese Datenstruktur aussehen soll, doch ist im Standard nicht vorgeschrieben, dass diese Datenstruktur auch in irgendeinem Header definiert ist. Aus diesem Grund verwaltet jede UNIX-Variante die unklare Spezifikation anders. Zum Beispiel definieren Solaris und Linux die Datenstruktur nicht, während LynxOS sie definiert. Deshalb befindet sich eine Definition der Datenstruktur *semun* im Beispielprogramm. Unter LynxOS darf die Datenstruktur nicht definiert werden. Dies wird durch den Präprozessor aber verhindert. Im obigen Beispiel wird nur eine Semaphore erzeugt. Für die Initialisierung der Semaphore wird dem Befehl *semctl* die Semaphoren-Id, *SETVAL* und die Datenstruktur *senum* übergeben. Mit *SETVAL* wird angegeben, dass die Semaphore auf einen bestimmten Wert, der in der Variable *val* von der Datenstruktur *senum* definiert ist, gesetzt wird. Mit *senctl* kann man auch die Semaphore wieder freigeben. Hierzu wird statt *SETVAL* die Option *IPC_RMID* dem Befehl übergeben.

Das Hoch- und Runterzählen der Semaphoren geschieht über den Befehl *semop*. Durch die Datenstruktur *sembuf* wird genau definiert, was der Befehl *semop* leisten soll. In der Variable *sem_num* der Datenstruktur *sembuf* wird angegeben, auf welche Semaphore sich die Datenstruktur bezieht. Da im obigen Beispiel nur eine Semaphore erzeugt wurde, kann hier Null eingetragen werden. In der Variablen *sem_op* wird der Wert eingetragen, der zu der Semaphore addiert werden soll. Soll die Semaphore um eins erhöht werden, so wird hier eine 1 eingetragen. Soll sie um eins erniedrigt werden, so muss eine -1 eingetragen werden. Die Variable *sem_flg* kann im Rahmen dieses Praktikums vernachlässigt werden. Somit wird hier eine Null eingetragen.

Das Problem der Prozesssynchronisation wurde gelöst, indem gewährleistet wird, dass nur ein Prozess exklusiven Zugriff auf die Variable *counter* hat. Hierzu wurde eine Semaphore verwendet, die zuvor auf eins initialisiert wurde. Bevor ein Prozess auf die Variable *counter* zugreift, erniedrigt er die Semaphore durch die Funktion *semaphore_wait* um eins. Falls der Wert der Semaphore Null ist, blockiert der Prozess, bis der Wert wieder erhöht wird. Somit kann nur ein Prozess auf die Variable *counter* zugreifen. Nachdem er die Variable *counter* erhöht hat, gibt er den Zugriff auf die Variable wieder frei, indem er die Semaphore durch die Funktion *semaphore_post* um eins erhöht.

## 7.6 Tipps und Tricks

Man muss bei der Verwendung von Semaphoren und gemeinsamem Speicher darauf achten, dass die Semaphoren bzw. Segmente beim Beenden des Programmes wieder freigegeben werden, denn das Betriebssystem stellt nur eine begrenzte Anzahl von Semaphoren und gemeinsamen Speichersegmenten zur Verfügung. Stürzt ein Programm ab, kann es sein, dass das Programm die Semaphoren bzw. gemeinsamen Speichersegmente nicht wieder freigibt. Das Kommando *ipcs* liefert eine Liste aller allokierten Semaphoren und gemeinsamen Speichersegmente. Z.B. sieht die Ausgabe unter Solaris wie folgt aus:

```
zarquon - /home/stefan> ipcs
IPC status from <running system> as of Thu Aug 12 11:23:53 MET DST
T         ID      KEY        MODE        OWNER    GROUP
Message Queues:
Shared Memory:
m          0   0x50000175 --rw-r--r--     root     root
Semaphores:
s          0   0x187cf    --ra-ra-ra-     root     root
```

Im obigen Beispiel besitzt der Systemadministrator ein gemeinsames Speichersegment mit der Id 0 und dem Schlüssel 0x50000175 sowie eine Semaphore mit der Id 0 und dem Schlüssel 0x187cf. Durch den Befehl *ipcrm* können Semaphoren bzw. Segmente gelöscht werden, die z.B. durch einen Programmabsturz übrig geblieben sind. Eine genaue Beschreibung des Befehls *ipcrm* erhält man durch den Befehl *man ipcrm*.

# 8 Die Thread-Programmierung

## 8.1 Prozesse und Threads

Moderne Betriebssysteme (und insbesondere Multiprozessor-Betriebssysteme) stellen eine fein gegliederte Prozessstruktur zur Verfügung. Ein Prozess besteht aus:

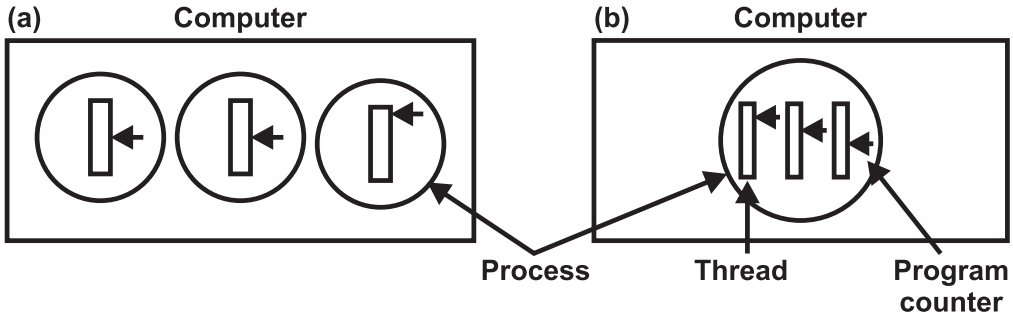
- einem (privaten) Adressraum für
  - den Programmcode und evtl.
  - Daten
- den Kontext-Daten des Prozesses (wie Seitentabellen für die MMU, Dateideskriptoren, Accounting-Informationen, ...)
- einer oder mehreren Aktivitätsbahnen, den sogenannten *Threads*, die Programmcode ausführen (mit den jeweils notwendigen Daten, wie Programmzähler, Stackpointer, Prozessorregisterinhalte).

Die wesentliche Erweiterung hierbei ist, dass es in einem einzelnen Prozess evtl. mehrere Aktivitätsbahnen nebeneinander geben kann. Alle diese Threads bewegen sich im gleichen Adressraum, sehen also alle dieselben Daten und haben auf alle Betriebsmittel des Prozesses Zugriff, wie Dateien, Signale, ..., die nicht einem einzelnen Thread, sondern dem Prozess als Ganzes zugeordnet sind (siehe Abbildung 8.1). In einem Multiprozessorsystem können mehrere Threads eines Prozesses unter Ausnutzung mehrerer Prozessoren gleichzeitig aktiv sein. Mehrere nebenläufige Aktivitätsbahnen innerhalb eines Adressraumes sind ein interessantes Modell für die parallele Programmierung. Ein weiterer Vorteil ergibt sich beim Wechsel der Zuteilung des Prozessors zu den Aktivitätsbahnen. Dieser Wechsel wird *Kontext-Wechsel* (context-switch) genannt, der zwischen zwei Threads desselben Prozesses wesentlich schneller als zwischen zwei Threads unterschiedlicher Prozesse ist. Viele der Arbeiten, die bei einer Prozessumschaltung auszuführen sind, sind beim Wechsel zwischen zwei Threads desselben Prozesses nicht notwendig. So z. B. das Einrichten der Seitentabellen des Adressraumes des neuen Prozesses. Dies ist eine willkommene Eigenschaft für z. B. Realzeitsysteme, die zeitkritisch zwischen einer Vielzahl unterschiedlicher Aufgaben wechseln oder schnell auf Unterbrechungen reagieren müssen. Beide Aspekte werden in den Vorlesungen Betriebssysteme II und Realzeitsysteme I vertieft.

Bei der Integration des Thread-Konzeptes in das System gibt es zwei unterschiedliche Ansätze, die in Abbildung 8.2 dargestellt sind:

- Die verschiedenen Threads eines Prozesses werden innerhalb des Benutzerprozesses verwaltet (sog. *User-Level Threads*). Das Betriebssystem hat keine Kenntnis von ihnen.
- Die verschiedenen Threads eines Prozesses sind dem Kern bekannt und werden von diesem verwaltet (sog. *Kernel-Threads* oder auch *leichtgewichtige Prozesse*).

User-Level Threads haben sich u. a. aufgrund der Historie von Betriebssystemen, die von Haus aus keine Threads kennen, sondern nur genau eine Aktivitätsbahn pro Pro-




**(a) Three processes with one thread each.**
**(b) One process with three threads.**

| Per thread items |
| --- |
| Program counter |
| Stack |
| Register set |
| Child threads |
| State |

| Per process items |
| --- |
| Address space |
| Global variables |
| Open files |
| Child processes |
| Timers |
| Signals |
| Semaphores |
| Accounting information |

**Abbildung 8.1:** Drei Prozesse mit je einem Thread, gegenüber einem Prozess mit drei Threads und den Daten, die zu verwalten sind.

zess, entwickelt. Um dennoch Threads benutzen zu können, werden diese innerhalb des Benutzerprozesses zur Verfügung gestellt.

Ein wesentlicher Unterschied zwischen User-Level und Kernel-Threads ist, dass letztere vom Betriebssystem auch bei der Prozessorzuteilung (dem Scheduling) als einzelne Aktivitätsbahnen berücksichtigt werden können. Sie können eigene angemessene Prioritäten und/oder Zeitscheiben haben. Dies trägt dazu bei, das Gesamtsystem möglichst effektiv und gerecht zu nutzen. User-Level Threads werden dagegen alle innerhalb des Prozessor-Kontingents des betreffenden Prozesses abgearbeitet. Nur auf dieser groben Granularität der Prozesse kann das Betriebssystem die Betriebsmittel (insbesondere den Prozessor) verwalten und verteilen. Bei User-Level-Threads findet ein zusätzliches

Scheduling auf Anwenderebene findet, dass einen Overhead darstellt.

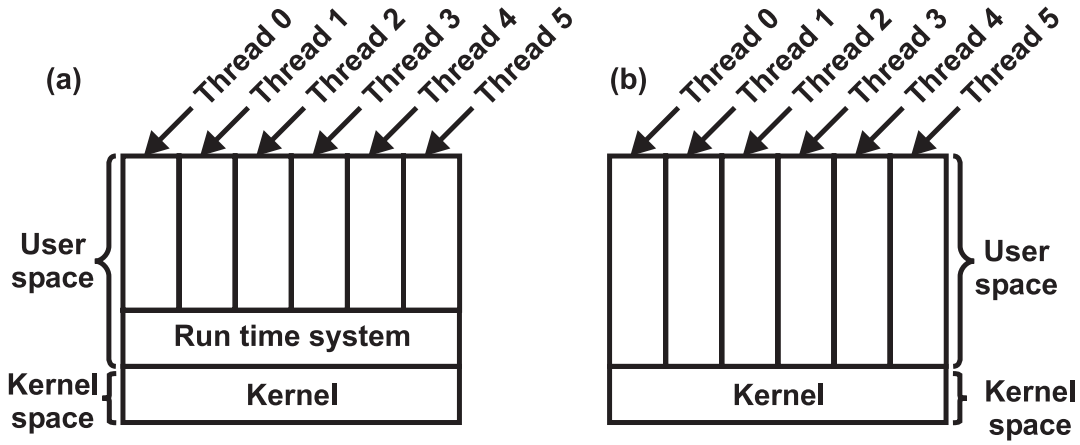



**Abbildung 8.2:** Zwei unterschiedliche Einordnungen von Threads in das System: (a) als User-Level Threads und (b) als Kernel-Threads.

## 8.2 Grundlagen für die Pthread-Programmierung (POSIX1.c)

Inzwischen wurde ein Standard für die Thread-Programmierung verabschiedet, der unter dem Namen *Pthreads (POSIX Threads)* bekannt ist. Inzwischen existiert auch eine Pthread-Implementierung für Linux, die unter *http://pauillac.inria.fr/~xleroy/linuxthreads* zu finden ist.

In diesem Praktikum wird natürlich keine umfassende Einführung in die Thread-Programmierung gegeben. Eine ausführliche Dokumentation der Pthreads findet man aber unter [7], [10] und [12]. Einige wichtige Voraussetzungen sollten bei der Entwicklung von Programmen mit Pthreads beachtet werden. In einem C-Programm sollte stets das Makro *_REENTRANT* definiert sein, da nur so gewährleistet werden kann, dass die threadsicheren Bibliotheken verwendet werden. Am einfachsten wird das Makro *_REENTRANT* durch die Compileroption *-D_REENTRANT* definiert. Die Prototypen der Pthread-Funktionen können durch *#include<pthread.h>* definiert werden. Außerdem muss dem Linker (Binder) mitgeteilt werden, dass er die Bibliothek *pthread* hinzubinden soll. Dies wird durch die Option *-lpthread* dem Linker mitgeteilt.

LynxOS ist ein Realzeit-Betriebssystem, das sehr stark die Thread-Programmierung einsetzt. In diesem Betriebssystem existieren sogar Threads auf Kernebene, d.h. Threads können sogar in der Entwicklung von Treibern eingesetzt werden. Aus diesem Grund hat die Firma Lynx, die LynxOS herstellt, den *gcc* erweitert. Wird ein Programm entwickelt, das Threads einsetzt, so sollte das Programm mit der Option *-mthreads* compiliert werden. Allerdings sollte beim Compilieren neben dem Makro *_REENTRANT* noch *_POSIX_THREADS_CALLS* definiert sein. Wird ein solches Programm mit der Option *-mthreads* gelinkt, so kann man die Option *-lpthread* weglassen.

QNX 6 (Neutrino) stellt inzwischen auch Kernel-level threads zur verfügung, welche dem POSIX Thread Standard folgen.

Bis der *PThreads-Standard* verabschiedet wurde, vergingen viele Jahre. (In den meisten Standardisierungsgremien wird heftig und ausgiebig diskutiert, so dass hier viel Zeit verloren geht. Es geht hier meistens nicht um fachliche Fragen, sondern firmenpolitische

Interessen verhindern oft eine Verabschiedung.) Einige Firmen gingen schon während der Standardisierungsphase mit einer eigenen *PThread-Implementierung* auf den Markt. Unter anderem hat Lynx ihr Betriebssystem in der verwendeten Version mit *PThreads* nach dem *PThreads-Standard Draft 4* ausgestattet. Leider stimmt dieser Entwurf nicht mit der endgültigen Fassung überein. Der Unterschied ist aber minimal. Da aber auch LynxOS in diesem Praktikum verwendet wird, werden neben den offiziellen Befehlen des *PThreads-Standard* auch die aus dem *Draft 4* erläutert. Die Abkürzung für den PThread-Standard lautet *POSIX.1c*. LynxOS soll in den neuen Versionen dazu kompatibel sein.

## 8.3 Starten und Beenden der Threads

Ein Thread kann durch den folgenden Befehl erzeugt werden:

**POSIX.1c:**

```
int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t
*attr, void * (*start_func)(void *), void *arg);
```

**POSIX.1c Draft 4:**

```
int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t
attr, void * (*start_func)(void *), void *arg);
```

Der erzeugte Thread startet seine Ausführung mit der Funktion *start_func*, die den Zeiger *arg* als Argument entgegen nimmt. Durch diesen Zeiger kann der Programmierer dem Thread alle wichtigen Informationen übergeben. Meistens werden die benötigten Informationen in einer Datenstruktur abgelegt und die Adresse der Datenstruktur der Funktion übergeben. Da den Threads der Aufbau der Datenstruktur bekannt ist, können sie nun auf die Informationen zugreifen. Der Zeiger *new_thread_ID* zeigt auf eine Variable vom Typ *pthread_t*. Dort wird beim Erzeugen des Threads eine systemweite eindeutige Identifikationsnummer abgelegt. Mit Hilfe dieser Nummer werden die einzelnen Threads unterschieden. Durch das Argument *attr* werden einige Attribute des zu erzeugenden Threads festgelegt. Bevor das Attribut *attr* verwendet wird, muss es durch den folgenden Befehl initialisiert werden:

**POSIX.1c:**

```
int pthread_attr_init(pthread_attr_t *attr);
```

**POSIX.1c Draft 4:**

```
int pthread_attr_create(pthread_attr_t *attr);
```

Diese Funktion initialisiert das Attribut *attr* mit den Standardwerten, so dass jetzt mit Hilfe dieses Attributs ein Thread erzeugt werden kann. Damit beim POSIX.1c Standard Threads auf mehrere Prozessoren verteilt werden, muss hier das zuvor initialisierte Attribut durch folgenden Befehl verändert werden.

```
int pthread_attr_setscope(pthread_attr_t *attr,
                          int *contentionscope);
```

In diesem Zusammenhang wird für das Argument *contentionscope* das im Header *pthread.h* definierte Makro *PTHREAD_SCOPE_SYSTEM* eingesetzt. Beim Entwurf des Standards können die Threads immer auf mehrere Prozessoren verteilt werden. Hier reicht es also aus, das Attribut *attr* mit den Standardwerten zu initialisieren.

Es existieren aber noch eine Reihe von Attributen, die beim Erzeugen eines Threads bestimmt werden können. Zum Beispiel kann die Größe des Stacks von einem Thread durch die folgende Funktion bestimmt werden:

**POSIX.1c und POSIX.1c Draft 4:**

```
int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t stacksize);
```

Meistens reicht es aber aus, den Standardwert für die Stackgröße zu verwenden. Somit wird der Befehl relativ selten verwendet. Wird das Attribut *attr* nicht mehr benötigt, so sollte dies dem Betriebssystem durch folgende Funktion mitgeteilt werden:

**POSIX.1c:**

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

**POSIX.1c Draft 4:**

```
int pthread_attr_delete(pthread_attr_t *attr);
```

Der Thread terminiert, sobald er die Funktion *start_func* verlassen hat. Die Funktion *start_func* stellt die main-Funktion des Threads dar, denn auch in einem normalen C-Programm terminiert das Programm, wenn die main-Funktion verlassen wird.

Mit dem Befehl

**POSIX.1c und POSIX.1c Draft 4:**

```
int pthread_join(pthread_t target_thread, void **status);
```

wartet der aufrufende Thread, bis der Thread mit der Identifikationsnummer *target_thread* terminiert ist. An der Speicheradresse, auf die die Variable *status* zeigt, wird der Rückgabewert des Threads geschrieben. Da für dieses Praktikum der Rückgabewert nicht interessant ist, wird für diese Variable *NULL* eingetragen. Das Beispielprogramm in Abbildung 8.3 soll nochmals die wichtigsten Funktionen erläutern. Durch den Präprozessor wird recht einfach zwischen den beiden Standards gewechselt. Wird das Programm auf einem LynxOS-Rechner compiliert, so wird der Entwurf verwendet, bei allen anderen Betriebssystemen (Linux und Solaris) wird der offizielle Standard verwendet. Leider sieht das Programm dadurch etwas wüst aus.

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

void* print_message_function(void *);

int main(int argc, char** argv)
{
    pthread_t thread1, thread2;
    pthread_attr_t attr;
    char* message1 = "Hello";
    char* message2 = "World";
```

```
#ifdef __Lynx__
    pthread_attr_create(&attr);
#else
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
#endif

#ifdef __Lynx__
    pthread_create(&thread1, attr, print_message_function,
    (void*) message1);
    pthread_create(&thread2, attr, print_message_function,
    (void*) message2);
#else
    pthread_create(&thread1, &attr, print_message_function,
    (void*) message1);
    pthread_create(&thread2, &attr, print_message_function,
    (void*) message2);
#endif

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

#ifdef __Lynx__
    pthread_attr_delete(&attr);
#else
    pthread_attr_destroy(&attr);
#endif

    printf("\n\nProgramm wird nun beendet\n\n");
    return 0;
}

void* print_message_function(void *ptr)
{
    char* message;
    message = (char *) ptr;
    printf("%s ", message);

    return(0);
}
```

**Abbildung 8.3:** Beispielprogramm zum Erzeugen von Threads

Das Programm erzeugt zwei Threads, die „Hello“ und „World“ auf den Bildschirm ausgeben und wartet solange, bis beide Threads terminiert sind. Anschließend gibt es noch „Programm wird nun beendet“ auf dem Bildschirm aus.

## 8.4 Threadsynchronisation

In einem Rechnersystem, in dem mehrere Threads asynchron (unabhängig voneinander) ablaufen, ist unter dem Aspekt des Zugriffs auf gemeinsame Daten und Betriebsmittel eine Threadsynchronisation erforderlich. Das Beispiel in Abbildung 8.4 zeigt, wie ohne eine solche Synchronisation bestimmte Operationen auf einem gemeinsamen Betriebsmittel, die gleichzeitig und unkontrolliert durch mehrere Threads erfolgen, zu falschen Ergebnissen führen können.

| **Thread A:** | **Thread B**: |
|---|---|
| finde freien Speicher in Liste fm;<br>freier Speicherbereich f; | |
| | finde freien Speicher in Liste fm;<br>freier Speicherbereich f; |
| markiere f als belegt;<br>beschreibe f ‘; | |
| | markiere f als belegt;<br>beschreibe f ‘; |

**Abbildung 8.4:** Beispiel für unkontrollierte Betriebsmittelvergabe

In dem Beispiel ist das gemeinsame Betriebsmittel, auf das von zwei Threads unkontrolliert zugegriffen wird, die Liste, die Informationen über belegten und freien Arbeitsspeicher enthält.

Durch den gemeinsamen unkontrollierten Zugriff auf diese Liste sind die Informationen in der Liste falsch. Der Fehler kann dadurch vermieden werden, dass allen anderen Threads ein Zugriff auf die Liste verwehrt wird, solange der Thread A in der Liste sucht und Eintragungen vornimmt.

Allgemein ist es also notwendig, Operationen von einzelnen Threads auf gemeinsame Betriebsmittel zu synchronisieren.

### 8.4.1 Der gegenseitige Ausschluss

Ein Verfahren zur Synchronisation ist der gegenseitige Ausschluss (mutual exclusion). Ein gegenseitiger Ausschluss garantiert dem Anwender, dass nur ein Thread auf die Datenstruktur (z.B. Zugriff auf eine Liste) zugreifen kann. Die Pthreads stellen sogenannte Mutexe (Abkürzung für mutual exclusion) zur Verfügung, die einen gegenseitigen Ausschluss garantieren.

Vor dem Zugriff auf die betreffende Datenstruktur wird durch das Betriebssystem ein Mutex angefordert. Das Betriebssystem blockiert den aufrufenden Thread, solange ein anderer Thread Zugriffserlaubnis besitzt und sperrt so die entsprechende Datenstruktur für alle anderen Threads. Die Anforderung eines Mutex erfolgt durch den Befehl:

**POSIX.1c und POSIX.1c Draft 4:**

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

Dabei spezifiziert die Variable *mp* den Mutex. Ist der Mutex bereits vergeben, so blockiert der Thread bis ein anderer Thread durch den Befehl

**POSIX.1c und POSIX.1c Draft 4:**

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

den Mutex wieder freigibt. Der Befehl

**POSIX.1c und POSIX.1c Draft 4:**

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

versucht ebenfalls, ein Mutex anzufordern. Besitzt bereits ein anderer Thread das Mutex, so blokkiert der Thread nicht, sondern liefert eine Fehlermeldung. Das heißt, der Rückgabewert der Funktion beträgt nicht Null. Bevor ein Mutex angefordert wird, muss dieses durch das Betriebssystem initialisiert werden. Dies kann durch den Befehl

**POSIX.1c und POSIX.1c Draft 4:**

```
int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t
*attr);
```

erreicht werden. Mit der Variablen *attr* werden einige Attribute des Mutexs spezifiziert. In diesem Praktikum reicht es aus, die Mutexe mit den Standardwerten zu initialisieren. Dazu wird die Variable *attr* wie folgt initialisiert:

**POSIX.1c:**

```
int pthread_mutexattr_init(&mutexattr);
```

**POSIX.1c Draft 4:**

```
int pthread_mutexattr_create(&mutexattr);
```

Jedes Betriebsmittel, das angefordert wurde, sollte ordnungsgemäß nach Gebrauch freigegeben werden. Bei einem Mutex wird dies durch den folgenden Befehl erreicht:

**POSIX.1c und POSIX.1c Draft 4:**

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

Als Beispiel für die Verwendung von Threads wird im nächsten Abschnitt das Problem „Dining Philosophers“ gelöst.

## 8.5 Beispiel: „Dining Philosophers“

Das Problem der speisenden Philosophen lässt sich wie folgt beschreiben: Fünf Philosophen sitzen an einem mit fünf Tellern und Gabeln gedeckten Tisch, wie in Abbildung 8.5 illustriert ist. Da auf dem Speiseplan glitschige Spaghetti stehen, benötigt jeder Philosoph zwei Gabeln zum Essen.

Das Leben eines Philosophen besteht stark reduziert aus alternierenden Folgen des Essens und Denkens. Wenn ein Philosoph hungrig wird, versucht er in beliebiger Rei-

henfolge seine linke und rechte Gabel aufzugreifen, isst dann eine bestimmte Zeit und legt die Gabeln wieder an ihren Platz zurück. Die Frage ist nun, ob für jeden Philosophen ein Algorithmus geschrieben werden kann, so dass die Philosophen sich nicht gegenseitig behindern? Ein Lösungsansatz für dieses Problem wird in der Abbildung 8.6 dargestellt.

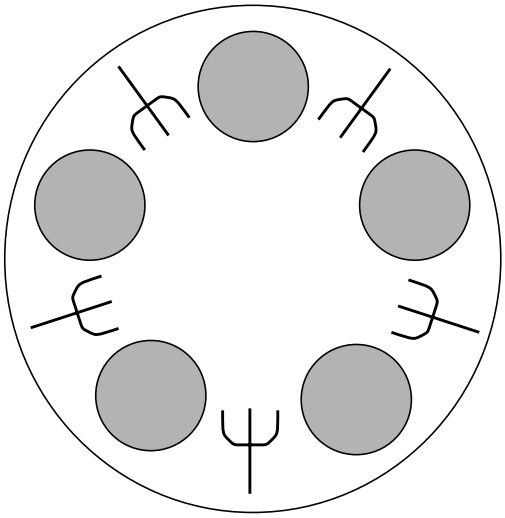


**Abbildung 8.5:** Essenszeit in der philosophischen Fakultät

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#ifdef __Lynx__
#include <time.h>

#else
#include <sys/time.h>
#endif
#include <pthread.h>

#define N          5                    /* #N Philosophen       */
pthread_mutex_t forks[N];               /* ein Mutex pro Gabel  */

void* philosopher(void*);
void  think(int);
void  eat(int);
void  take_fork(int, int);
void  put_fork(int, int);

void* philosopher(void* j)
{
    int x;
    int i = *((int*) j);          /* Philosoph Nummer 0-N-1 */
```

```
    for(x=0; x<10000; x++)
    {
        think(i);/* Philosoph denkt */
        printf("Der %i. Philosoph moechte essen.\n", i);
        take_fork(i, i); /* Philosoph nimmt die linke Gabeln auf
                            oder blockiert  */
        take_fork(i, (i+1) % N);  /*  Philosoph nimmt die rechte
                                      Gabeln auf oder blockiert */
        eat(i);                   /*  Philosoph isst Spaghetti  */
        put_fork(i, i);           /*  Philosoph legt die linke
                                      Gabel zurueck             */
        put_fork(i, (i+1) % N);   /*  Philosoph legt die rechte
                                      Gabel zurueck             */
    }
    return(0);
}

void take_fork(int phil, int i)
/* Der Philosoph moechte die Gabel i nehmen. */
{
    pthread_mutex_lock(&forks[i]);
    printf("Der %i. Philosoph hat die Gabel %i genommen.\n",
           phil, i);
}

void put_fork(int phil, int i)
/* Der Philosoph legt die Gabel i wieder zurueck. */

{
    pthread_mutex_unlock(&forks[i]);
    printf("Der %i. Philosoph hat die Gabel %i zurueck gelegt\n",
           phil, i);
}

void think(int phil)
{
    unsigned int i;
    double x = 10.0;

    printf("Der %i. Philosoph denkt.\n", phil);
    for(i=0; i<rand(); i++)/*Diese Schleife simuliert das Denken.*/
        x += log10(x);
}

void eat(int phil)
{
```

```
    unsigned int i;
    double x = 10.0;
    printf("Der %i. Philosoph isst.\n", phil);
    for(i=0; i<rand(); i++)/*Diese Schleife simuliert das Essen.*/
        x += log10(x);
}

int main(int argc, char** argv)
{
    int i, j[N];
    struct timeval tp;
    pthread_t id[N];
    pthread_attr_t attr;
    pthread_mutexattr_t mutexattr;

#ifdef __Lynx__
    pthread_mutexattr_create(&mutexattr);
#else
    pthread_mutexattr_init(&mutexattr);
#endif

    gettimeofday(&tp, NULL);
    srand(tp.tv_usec); /*  Intialisiert den Zufallsgenerator   */
    for(i=0; i<N; i++) /*  In dieser Schleife werden die Mutexe
                           initialisiert.                      */
#ifdef __Lynx__
        pthread_mutex_init(&forks[i], mutexattr);

#else
        pthread_mutex_init(&forks[i], &mutexattr);
#endif

#ifdef __Lynx__
    pthread_attr_create(&attr); /*  Initializiert die Attribute
                                    der Threads */
#else
    pthread_attr_init(&attr);   /*  Initializiert die Attribute
                                    der Threads */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* Diese Attribut wird gesetzt, damit die Threads auf   */
    /* mehreren Prozessoren verteilt werden.                */
#endif

    for(i=0; i<N-1; i++)
    /* In dieser Schleife werden die Threads erzeugt. */
    {
```

```
        j[i] = i;
#ifdef __Lynx__
        pthread_create(&id[i], attr, philosopher, (void*) &j[i]);
#else
        pthread_create(&id[i], &attr, philosopher, (void*) &j[i]);
#endif
    }

    j[N-1] = N-1;
    philosopher((void*) &(j[i]));
    for(i=0; i<N-1; i++)  /*  Der Hauptthread wartet bis alle
                              anderen Threads terminiert sind. */
        pthread_join(id[i], NULL);

    for(i=0; i<N; i++)  /*  In dieser Schleife werden die Mutexe
                            wieder frei gegeben. */
        pthread_mutex_destroy(&forks[i]);

#ifdef __Lynx__
    pthread_attr_delete(&attr);
#else
    pthread_attr_destroy(&attr);
#endif
    return 0;
}
```

**Abbildung 8.6:** Eine falsche Lösung des Problems der speisenden Philosophen

Dieser Lösungsansatz ist relativ einfach. Da jeder Philosoph nur eine Gabel besitzen kann, werden sie durch die Mutexe *forks* simuliert. Möchte ein Philosoph essen, so versucht er zunächst, die linke Gabel und danach die rechte Gabel zu nehmen. Ist zum Beispiel die linke Gabel vergeben, so wird der Philosoph durch das Mutex blockiert, bis die Gabel wieder zur Verfügung steht. Leider arbeitet der Algorithmus nicht korrekt. Angenommen, alle Philosophen wollen gleichzeitig essen, so nehmen sie gleichzeitig die linke Gabel auf. Danach wollen sie die rechte Gabel aufnehmen, doch diese besitzt bereits der rechte Nachbar. Nun warten alle Philosophen auf den rechten Nachbarn, bis dieser seine Gabel zurücklegt. Es ist also ein *zirkuläres Warten* entstanden, bei dem jeder Philosoph auf seinen rechten Nachbarn wartet. Kein Philosoph kann nun seine Spaghettis essen. Somit müssen alle Philosophen verhungern. Eine solche Situation wird *Deadlock* genannt. Der Programmierer sollte unbedingt darauf achten, dass kein zirkuläres Warten in seinem Programm entstehen kann.

# 9 Die Echtzeiterweiterung POSIX.4

Wie im Kapitel 3 bereits erläutert, eignet sich UNIX eigentlich nicht als Realzeit-Betriebssystem. Da UNIX aber weit verbreitet ist, haben einige Firmen versucht, UNIX zu erweitern, so dass es Echtzeitfähigkeiten besitzt. Zur Zeit existieren Bestrebungen, die vielen propitären Lösungen durch einen Standard zu ersetzen. Dieser Standard ist unter dem Namen POSIX.4 (offiziell POSIX.1b) bekannt. Solaris unterstützt zum Beispiel diesen Standard, obwohl nicht behauptet werden kann, dass es sich bei Solaris um ein Realzeit-Betriebssystem handelt. Ein realzeitfähiges Betriebssystem, das den POSIX.4 Standard unterstützt, ist LynxOS. Leider unterstützt Linux diesen Standard nur teilweise. In den folgenden Abschnitten soll eine kleine Einführung in den POSIX.4 Standard gegeben werden. Unter Solaris muss bei der Verwendung von POSIX.4 darauf geachtet werden, dass die Bibliothek *libposix4.a* hinzu gebunden wird. Dies geschieht am einfachsten, wenn beim Linken die Option *-lposix4* gesetzt wird.

## 9.1 Signals unter POSIX.4

Das Signal-Handling aus Abschnitt 7 eignet sich nicht für realzeitfähige Anwendungen. Die bestehenden Probleme können durch folgende Stichpunkte zusammengefasst werden:

- Es existieren zu wenig Signals und diese bieten zu wenig Informationen an.
- Normale Signals werden meistens durch ein Feld von Bits verwaltet. Wird ein Signal ausgelöst, so wird ein bestimmtes Bit im Feld gesetzt und nach dem Behandeln des Signals wieder gelöscht. Wird aber ein Signal ausgelöst, dessen Bit bereits gesetzt ist, geht dieses Signal verloren.
- In einem normalen UNIX-System besteht keine feste Reihenfolge, in der die Signals abgearbeitet werden, falls mehrere Signale ausgelöst wurden. Somit besteht die Gefahr der Prioritäten-Inversion.

POSIX.4 bietet neben den alten Signals auch weitere an, die der Anwender selber definieren kann. Dabei muss sich die Signal-Nummer zwischen den Konstanten *SIGRTMIN* und *SIGRTMAX* befinden. Das Signal mit der kleinsten Nummer wird zuerst abgearbeitet. Aus diesem Grund bekommen Signale mit hoher Priorität eine kleine Zahl. Das Signal mit der Nummer *SIGRTMIN* besitzt also die höchste Priorität, während das Signal mit der Nummer *SIGRTMAX* die kleinste Priorität besitzt. Die Reihenfolge der Abarbeitung der Standard-Signale unter UNIX wurde in POSIX.4 nicht näher spezifiziert. Die Standard-Signale werden aber meistens den POSIX.4-Signalen vorgezogen. Abbildung 9.1 soll die Verwendung von POSIX.4 Signalen verdeutlichen.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
```

```
#define N    10


/*************************************************************/
/* signal handler                                            */
/*************************************************************/
void ExitHandler(int sig)
{
    fprintf(stderr, "Programm wird beendet!!!\n");
    exit(0);
}

void RTMinHandler(int signum, siginfo_t* extra, void* v)
{
    static int i = 0;

    fprintf(stderr, "%i. Iteration Hello world !!! signum=%i
                     sival_int=%i\n", ++i, signum,
                     extra->si_value.sival_int);
}

int main(int argc, char** argv)
{
    struct sigaction     sINT, sTERM, sRTMIN;
    pid_t                pid;
    union sigval         sval;
    int                  i;

    /* define action for SIGINT */
    sINT.sa_handler = ExitHandler;
    sINT.sa_flags = 0;
    if (sigaction(SIGINT, &sINT, NULL) < 0)
    {
        perror("sigaction");
        exit(1);
    }

    /* define action for SIGTERM */
    sTERM.sa_handler = ExitHandler;
    sTERM.sa_flags = 0;
    if (sigaction(SIGTERM, &sTERM, NULL) < 0)
    {
        perror("sigaction");
        exit(1);
    }
```

```
    sRTMIN.sa_sigaction = RTMinHandler;
    sRTMIN.sa_flags = SA_SIGINFO; /* This is a queued signal */

    if (sigaction(SIGRTMIN, &sRTMIN, NULL) < 0)
    {
          perror("sigaction");
          exit(1);
    }

    pid = fork();

    if (pid == 0)
    {
          /* child process */
          while(1)
          sleep(1);
    } else {
            /* parent process */
        for(i=0; i<N; i++)
        {
            /* initialize additinal information */
            sval.sival_int = i;

            /* send signal with the id SIGRTMIN */
            if (sigqueue(pid, SIGRTMIN, sval) < 0)
                perror("sigqueue");
        }

        sleep(2);

        /* terminate child process */
        kill(pid, SIGTERM);
    }

    wait(NULL);

    fprintf(stderr, "Programm wird beendet!!!\n");

    return 0;
}
```

**Abbildung 9.1:** Beispielprogramm für die Verwendung von POSIX4. Signale

Mit der Funktion *sigaction* wird, wie im Abschnitt 7 beschrieben, der Signal-Handler für verschiedene Signale bestimmt. Für POSIX.4 Signale trägt man die Handler-Funktion nicht in die Variable *sa_handler* der Datenstruktur *sigaction* ein, sondern in *sa_sigaction*.

Durch den Eintrag *SA_SIGINFO* als Flag der Datenstruktur *sigaction*, werden die Signale vom Programm in einer Queue verwaltet. Dadurch können, wie oben beschrieben, keine Signale verloren gehen.

Normalerweise wird der Befehl *kill* verwendet, um ein bestimmtes Signal an einen Prozess zu schicken. Unter POSIX.4 wird hierzu *sigqueue* verwendet. Mit dem Befehl *sigqueue* können dem Signal-Handler noch weitere Informationen gesendet werden. Hierbei wird die Datenstruktur *sigval* dem Signal-Handler übermittelt. Sie wird ihm nicht direkt übergeben, sondern versteckt sich in der Datenstruktur *siginfo_t*. In *sigval* existiert zum Beispiel die Integer-Zahl *sival_int*, die gesetzt wurde, bevor das Signal durch *sigqueue* ausgelöst wurde. Neben *sigval* muss dem Befehl *sigqueue* noch die Prozess-Id des Prozesses, der das Signal erhalten soll, und die Nummer des Signals übermittelt werden. Mit dem Befehl *kill* wird ein normales UNIX-Signal ausgelöst. Hier fällt also die zusätzliche Information *sigval* weg, da dem Prozess bei normalen UNIX-Signalen keine weiteren Informationen übermittelt werden.

## 9.2 Periodische Timer unter POSIX.4

Ein periodische Timer wurde bereits im Abschnitt 7.2 erläutert. Dieser ist leider nicht sehr genau. Unter POSIX.4 kann mit *timer_create* ein Timer angefordert werden, der wesentlich genauer ist. Er bietet eine Auflösung im Nanosekundenbereich an. Bevor ein Programm terminiert, das einen Timer mit *timer_create* angefordert hat, sollte es den Timer durch *timer_delete* wieder freigeben. Der Startzeitpunkt und das Intervall kann durch die Funktion *timer_settime* gesetzt werden. Hierzu wird die Datenstruktur *itimerspec* benötigt, die sehr der Datenstruktur *itimerval* aus Abschnitt 7.2 ähnelt. Im Prinzip besteht der Unterschied nur darin, dass in der Datenstruktur *itimerspec* statt Sekunden und Mikrosekunden der Startzeitpunkt und das Intervall in Sekunden und Nanosekunden angegeben wird. Abbildung 9.2 zeigt ein Beispiel für die Verwendung des periodischen Timers.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>

timer_t              tid;

void ExitHandler(int sig)
{
     /* delete timer */
     timer_delete(tid);

     /* exit process */
     exit(0);
}
```

```
int main(int argc, char** argv)
{
    struct itimerspec    new_setting, old_setting;
    sigset_t             look_for_these;
    siginfo_t            extra;

    struct sigaction     sINT, sTERM;
    int                  i;

    /* define action for SIGINT */
    sINT.sa_handler = ExitHandler;
    sINT.sa_flags = 0;
    if( sigaction( SIGINT, &sINT, NULL) < 0 )
    {
        fprintf(stderr, "Error: sigaction\n");
        exit(1);
    }

    /* define action for SIGTERM */
    sTERM.sa_handler = ExitHandler;
    sTERM.sa_flags = 0;
    if( sigaction( SIGTERM, &sTERM, NULL) < 0 )
    {
        fprintf(stderr, "Error: sigaction\n");
        exit(1);
    }

    /* intialize sigset_t */
    sigemptyset(&look_for_these);
    sigaddset(&look_for_these, SIGALRM);
    sigprocmask(SIG_BLOCK, &look_for_these, NULL);

    /* create timer */
    i = timer_create(CLOCK_REALTIME, NULL, &tid);
    if (i < 0)
    {
        perror("timer_create");
        exit(1);
    }

    /* set timer values */
    new_setting.it_value.tv_sec = 1;
    new_setting.it_value.tv_nsec = 0;
    new_setting.it_interval.tv_sec = 0;
    new_setting.it_interval.tv_nsec = 500000000;
```

```
    /* start timer */
    i = timer_settime(tid, 0, &new_setting, &old_setting);
    if (i < 0)
    {
        perror("timer_settime");

        exit(1);
    }

    for(i=0; i<100; i++)
    {
        /* wait for timer expiration */
        if (sigwaitinfo(&look_for_these, &extra) < 0)
            perror("sigwaitinfo");
          else
            fprintf(stderr, "%i. Iterationsschritt\n", i);
    }

    /* delete timer */
    timer_delete(tid);

    exit(0);
}
```

**Abbildung 9.2:** Beispiel für die Verwendung eines POSIX.4 Timers

Der POSIX.4 Timer kann leider noch nicht unter Linux verwendet werden. Im obigen Beispiel soll das Programm nur auf das Signal des periodischen Timers warten und anschließend die Berechnung fortsetzen. In diesem Fall würde der Signal-Handler eine leere Funktion darstellen. Der Handler würde unnötig viel Zeit kosten, da das Programm unnötigerweise zum Handler und wieder zurück springen müsste. Deshalb kann in POSIX.4 mit der Funktion *sigwaitinfo* auf ein Signal gewartet werden, ohne den Signal-Handler aufzurufen. Hierzu muss die Datenstruktur *sigset_t* initialisiert werden. Zunächst wird mit Hilfe von *sigemptyset* die Datenstruktur auf die Standard-Werte gesetzt. Anschließend wird mit *sigaddset* mitgeteilt, dass mit Hilfe von *sigwaitinfo* auf ein Signal vom Typ *SIGALRM* gewartet wird, dass von einem periodischen Timer ausgelöst wird. Da kein Signal-Handler aufgerufen wird, soll das ankommende Signal (hier *SIGALRM*) so lange blockieren, bis es mit *sigwaitinfo* bearbeitet wurde. Um diese Eigenschaft des Signals zu erhalten, wird mit *sigprocmask* das Flag auf *SIG_BLOCK* gesetzt.

## 9.3 POSIX.4 Semaphoren

Wie bereits in Abschnitt 7.5 erwähnt, existiert im POSIX.4-Standard eine weitere Implementierung der bekannten Semaphoren. Die Lösung aus Abbildung 7.4 kann auch mit Hilfe der POSIX.4 Semaphoren wie folgt gelöst werden:

```
#include <unistd.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <semaphore.h>
#ifndef __Lynx__
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/fcntl.h>

#else
#include <ipc.h>
#include <shm.h>
#include <sem.h>
#include <fcntl.h>
#endif
#include <sys/types.h>
#include <sys/wait.h>

const int SIZE = 4*1024;          /* size of shared segment */
const int KEY = 0xFEDCBA;

int main(int argc, char* argv[])
{
    int shmid, pid, i = 0;
    sem_t semid;
    volatile   int* counter;

    /* create shared segment */
    shmid = shmget(KEY, SIZE*sizeof(int), O_RDWR|IPC_CREAT|511);
    if (shmid == -1)
    {
        perror("shmget");
        exit(1);
    }

    /* map segement into the virtual address space */
    if ((int) (counter = (int*) shmat(shmid, 0, O_RDWR)) == -1)
    {
        perror("shmat");
        exit(1);
    }

    /* initialize counter */
    *counter = 0;

    /* create semaphore */
    if (sem_init(&semid, 1, 1) == -1)
```

```
    {
        perror("sem_init");
        exit(1);
    }

    /* create new child process */
    pid = fork();

    while(i < 50)
    {
        sem_wait(&semid);
        fprintf(stderr, "pid = %ii = %i*counter = %i\n",
               (int) getpid(), i+1, *counter + 1);
        (*counter)++;
        i++;
        sem_post(&semid);
    }

    /* parent wait for a child process to terminate */
    wait(NULL);

    if (pid != 0)
    {
        /* remove semaphore */
        sem_destroy(&semid);

        /* unmap shared segment */
        shmdt((void*) counter);

        /* remove shared segment  */
        shmctl(shmid, IPC_RMID, 0);
    }

    return(0);
}
```

**Abbildung 9.3:** Prozesssynchronisation durch POSIX.4 Semaphoren

Durch den Befehl

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

wird eine Semaphore erzeugt. Der Befehl trägt an der Speicheradresse *sem* die Identifikationsnummer der Semaphore ein. Mit der Variablen *pshared* kann angegeben werden, ob nur Threads oder auch Prozesse über diese Semaphore synchronisiert werden. Sollen nur Threads synchronisiert werden, so wird hier eine Null eingetragen, anderenfalls ein Wert ungleich Null. Linux unterstützt bei seiner Implementierung der POSIX.4 Semaphoren nicht die Prozesssynchronisation, daher funktioniert das obige Beispiel leider

nicht unter Linux. Bei anderen Betriebssystemen, wie z.B. Solaris und LynxOS, stellt dies kein Problem dar. Durch die Variable *value* wird der Wert, auf dem die Semaphore beim Erzeugen initialisiert wird, bestimmt. Durch den Befehl *sem_post* wird die Semaphore um eins erhöht, während der Befehl *sem_wait* ihn um eins erniedrigt. Wie bei den Semaphoren aus Abschnitt 7.5 blockieren die Prozesse bzw. Threads, wenn versucht wird, durch *sem_wait* eine Semaphore zu erniedrigen, die den Wert Null besitzt. Diese Blockade wird erst dann aufgelöst, wenn ein anderer Prozess bzw. Thread die Semaphore durch *sem_post* erhöht. Durch den Befehl *sem_destroy* werden die Ressourcen der Semaphore wieder freigegeben.

Es existiert noch eine zweite Möglichkeit, eine POSIX.4 Semaphore zu erzeugen: Angenommen, es sollen zwei vollkommen verschiedene Prozesse, die nicht mit dem Befehl *fork* voneinander abgespaltet werden, über diese Semaphore synchronisiert werden, so entsteht bei der Erzeugung über den Befehl *sem_init* das Problem, dass die Semaphoren-Id ausgetauscht werden muss. Dies könnte man über ein gemeinsames Speicherstück erreichen. Als Alternative stehen die Befehle *sem_open* und *sem_close* zur Verfügung. Hierbei können mehrere Prozesse über einen Schlüssel in Form eines Namens die gleiche Semaphore erzeugen. Aus diesem Grund werden solche Semaphoren *„named semaphores“* genannt, während die Semaphoren aus dem obigen Beispiel *„unnamed semaphores“* heißen. Nähere Erläuterungen zu den Befehlen *sem_open* und *sem_close* sind in den Manpages zu finden.

## 9.4 Manipulation der Scheduling-Eigenschaft

Zur Laufzeit kann die Priorität eines Prozesses abgefragt und gegebenenfalls geändert werden. Dabei muss man wissen, dass unter UNIX drei verschiedene Scheduling-Klassen definiert sind, wobei zwei Realzeit-Anwendungen vorbehalten sind:

- SCHED_FIFO: Das Scheduling-Verfahren nach dem First-In-First-Out-Prinzip ist das einfachste zu implementierende Scheduling-Verfahren. Der Prozess mit der höchsten Priorität bekommt in diesem Verfahren Rechenzeit, bis er terminiert oder ein anderer Prozess mit höherer Priorität rechenbereit ist. Dieses Scheduling-Verfahren ist Realzeit-Anwendungen vorbehalten.

- SCHED_RR: Ein weiteres Verfahren für Realzeit-Anwendungen stellt das Round-Robin-Verfahren dar, das ausführlich in [1] beschrieben wird.

- SCHED_OTHER: Das dritte Verfahren kann sich jedes UNIX-System selber aussuchen und definieren. Da UNIX normalerweise ein Mehrbenutzer-Betriebssystem ist, handelt es sich meistens um ein Verfahren, das Job-Fairness garantiert. Dieses Verfahren stellt das Standard-Verfahren eines UNIX-Prozesses dar.

Die beiden Scheduling-Verfahren, die speziell für Realzeit-Anwendungen entwickelt wurden, werden gegenüber dem dritten Verfahren bevorzugt. D.h., existiert im System ein rechenbereiter Prozess, der entweder Round-Robin- oder das FIFO-Verfahren verwendet, so erhalten alle Prozesse, die das Standard-Verfahren verwenden, keine Rechenzeit.

Bei einem UNIX-Prozess wird standardmäßig das dritte Verfahren mit niedrigster Priorität gestartet. Das verwendete Verfahren und die Priorität des Prozesses kann aber auch

zur Laufzeit bestimmt werden. Um das verwendete Scheduling-Verfahren zu bestimmen, wird der folgende Befehl verwendet:

```
int sched_getscheduler(pid_t pid);
```

Dieser Befehl ermittelt das Scheduling-Verfahren des Prozesses mit der Prozess-Id *pid*. Wird für die Prozess-Id eine Null eingetragen, so ermittelt der Befehl das Scheduling-Verfahren für den aufrufenden Prozess. Dieser Befehl liefert drei verschiedene Werte zurück, die die einzelnen Scheduling-Verfahren beschreiben. Diese Werte müssen nicht auswendig gelernt werden, da hierfür Makros definiert sind, die sich jeder leicht merken kann. Die Makros lauten:

- *SCHED_FIFO*
- *SCHED_RR*
- *SCHED_OTHER*

*SCHED_FIFO* steht für das FIFO-Verfahren, während *SCHED_RR* für das Round-Robin-Verfahren steht. Demzufolge steht *SCHED_OTHER* für das Scheduling-Verfahren, das standardmäßig verwendet wird. Die Priorität eines Prozesses wird durch den folgenden Befehl bestimmt:

```
int sched_getparam(pid_t pid, struct sched_param *p);
```

Dieser Befehl ermittelt die Parameter des aktuellen Scheduling-Verfahrens des Prozesses mit der Prozess-Id *pid*. Wird für die Prozess-Id eine Null eingetragen, so ermittelt der Befehl die Parameter für den aufrufenden Prozess. Die Parameter des aktuellen Scheduling-Verfahrens werden in der Datenstruktur *sched_param* abgelegt. Diese Datenstruktur enthält unter anderem die Variable *sched_priortity*, die die aktuelle Priorität des Prozesses enthält. Mit dem folgenden Befehl kann das Scheduling-Verfahren verändert werden:

```
int sched_setscheduler(pid_t pid, int policy, struct sched_param *p);
```

Wie bei den vorherigen Befehlen wird mit Hilfe der Prozess-Id *pid* der Prozess bestimmt, dessen Scheduler-Eigenschaften verändert werden sollen. Dabei steht Null auch hier für den aufrufenden Prozess. Für die Integer-Zahl *policy* wird die Art des Scheduler eingetragen, der verwendet werden soll. Hier können die Makros *SCHED_FIFO*, *SCHED_RR* oder *SCHED_OTHER* eingetragen werden. Durch die Datenstruktur *sched_param* werden die Parameter des Schedulers eingestellt. Z.B. wird hier die Priorität in die Variable *sched_priority* eingetragen. Doch um welche Werte handelt es sich hierbei? Durch die folgenden Befehle wird die obere und untere Grenze ermittelt, die in die Variable *sched_priority* eingetragen werden darf:

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

Die verschiedenen Scheduling-Verfahren können unterschiedlich viele Prioritätsstufen verwalten. Aus diesem Grund muss den Befehlen noch die Art des Scheduling-Verfahrens mitgeteilt werden.

In vielen Betriebssystemen (z.B. Solaris, Linux) benötigen die Befehle, die die Priorität eines Prozesses verändern, Administrator-Rechte. Alle hier verwendeten Befehle und Makros sind im Header *sched.h* definiert.

# 10Leistungsvergleich von Realzeitsyste-men

## 10.1 Begriffe und Definitionen

Es existieren viele Benchmarks, um Betriebssysteme zu evaluieren. Bei der Evaluation von Realzeit-Betriebssystemen stellt die *Interrupt Latency* eine wichtige Größe dar. Unter dem Begriff *Interrupt Latency* wird die Zeit verstanden, die zwischen dem Auslösen eines Interrupts und der Ausführung des ersten Befehls im zugehörigen Interrupt-Handler vergeht. Oft wird die *Interrupt Dispatch Latency* unterschlagen, die aber mindestens genauso wichtig ist. Diese Größe beschreibt die Zeit, die zwischen der Ausführung des letzten Befehls im Interrupt-Handler und der Fortsetzung des Programmes, das durch den Interrupt unterbrochen wurde, vergeht. Unter *Task Response Time* wird die Summe aus *Interrupt Latency, Interrupt Dispatch Latency* sowie die Dauer der eigentlichen Abarbeitung des Interrupts verstanden. Abbildung 10.1 soll diese Größen noch-

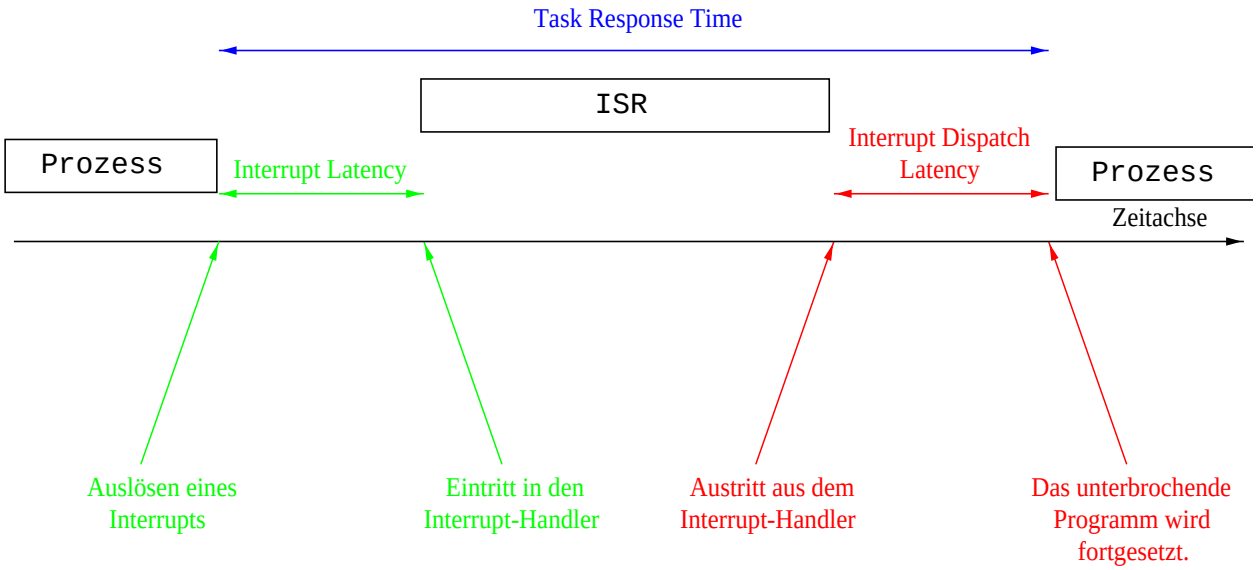



**Abbildung 10.1:** Anschauliche Erläuterung der Begriffe Interrupt Latency, Interrupt Dispatch Latency sowei Task Response Time

mals verdeutlichen. In dem nun folgenden Abschnitt wird eine Möglichkeit zur Messung dieser Zeiten vorgestellt und erläutert. Leider wird für diese Messung unter Linux und LynxOS ein Treiber benötigt. Daher wird in diesem Skript auf die Treiberentwicklung unter Linux und LynxOS eingegangen, was sich aber als nicht allzu schwierig herausstellen wird.

## 10.2 Programmierung von Treibern unter UNIX

Grundsätzlich gibt es zwei Arten von Geräten, block- und zeichenorientierte. Als blockorientierte Geräte werden solche bezeichnet, auf die man wahlfreien Zugriff hat, d.h. von denen beliebige Blöcke gelesen und geschrieben werden können. Für Dateisysteme ist der wahlfreie Zugriff unbedingt erforderlich. Sie können deshalb nur auf blockorientierten

Geräten untergebracht werden. Zeichenorientierte Geräte wiederum sind Geräte, die meist nur sequentiell arbeiten. Somit wird auf diese Geräte ungepuffert zugegriffen. In diese Klasse fällt die gebräuchlichste Hardware, wie Soundkarten, Scanner, Drucker usw., auch wenn sie intern mit Blöcken arbeiten. Diese Blöcke sind jedoch sequentieller Natur, da auf diese nicht wahlfrei zugegriffen wird.

Da im UNIX-Kern mehrere Gerätetreiber nebeneinander existieren müssen, werden sie anhand der *Major-Nummer* eindeutig identifiziert. Ein Gerätetreiber kann mehrere physische und virtuelle Geräte, z.B. mehrere Festplatten und Partitionen, verwalten. Deshalb wird das einzelne Gerät durch die *Minor-Nummer*, eine Zahl zwischen 0 und 255, angesprochen. Ein einzelnes Gerät ist demnach eindeutig durch den Gerätetyp (block- oder zeichenorientierte Geräte), die Major-Nummer des Gerätetreibers und seine Minor-Nummer identifiziert. Um das Gerät anzusprechen, wird im Verzeichnis */dev* eine Datei angelegt, die das Gerät repräsentiert. Mit *ls -la /dev* werden alle Geräte des Systems aufgelistet. Hier wird auch aufgelistet, welche Major- bzw. Minor-Nummer für ein einzelnes Gerät verwendet wird.

In jedem Treiber existieren fest definierte Einstiegspunkte. Nur über diese Punkte kann ein Programm mit dem Treiber kommunizieren. Je nachdem, welche Funktion verwendet wird, um mit dem Treiber zu kommunizieren, wird eine bestimmte Funktion (Einstiegspunkt) des Treibers aufgerufen. Mit dem Treiber wird u.a. über folgende Funktionen kommuniziert:

- *open* (Öffnen des Treibers)
- *close* (Schließen des Treibers)
- *read* (Liest Daten vom Gerät ein)
- *write* (Schickt Daten zum Gerät)
- *ioctl* (Sonder/Kontrollfunktionen)

Obwohl ein Gerät versucht, die Bedienung vom Gerät nach außen hin möglichst zu abstrahieren, hat doch jedes seine speziellen Eigenschaften. Dazu können verschiedene Operationsmodi ebenso wie gewisse Grundeinstellungen gehören. Auch eine Einstellung von Geräteparameter zur Laufzeit, wie IRQs, I/O-Adresse usw., ist denkbar. Um diese Grundeinstellungen zu verändern, wird meistens die *ioctl*-Funktion verwendet. Diese Funktion sieht unter UNIX wie folgt aus:

```
int ioctl(int fd, int request, char* arg);
```

Neben dem Filedeskriptor *fd*, den man beim Öffnen des Gerätes durch den Befehl *open* erhält, bekommt die Funktion eine Integer-Zahl und einen Zeiger, der auf Daten im Nutzeradressraum zeigt. Durch die Zahl *request* wird beschrieben, welche Eigenschaften des Treibers eingestellt werden. Der Zeiger *arg* zeigt auf die Daten, die der Befehl *ioctl* benötigt. Der *ioctl*-Befehl kann aber auch Daten an die Stelle kopieren, auf die der Zeiger zeigt.

## 10.3 Messung der Interrupt-Latenzzeit unter Linux und LynxOS

In diesem Praktikum soll die *Interrupt Latency* sowie die *Interrupt Dispatch Latency* gemessen werden. Es existieren viele Möglichkeiten, diese Zeiten zu messen. Mit Hilfe des Parallelport werden in diesem Praktikum diese Zeiten ermittelt. Möchte ein externes Gerät Daten zum Computer senden, so setzt dieses Gerät die entsprechenden Datenleitungen sowie die Acknowledge-Leitung des Parallelports. Die Acknowledge-Leitung zeigt dem Computer an, dass Daten auf dem Parallelport anliegen. Sobald die Acknowledge-Leitung gesetzt ist, löst der Parallelport einen Interrupt aus. Um die *Interrupt Latency* bzw. *Interrupt Dispatch Latency* zu messen, wird die Datenleitung 0 mit der Acknowledge-Leitung durch ein externes Gerät kurzgeschlossen (siehe Abbildung 10.2).

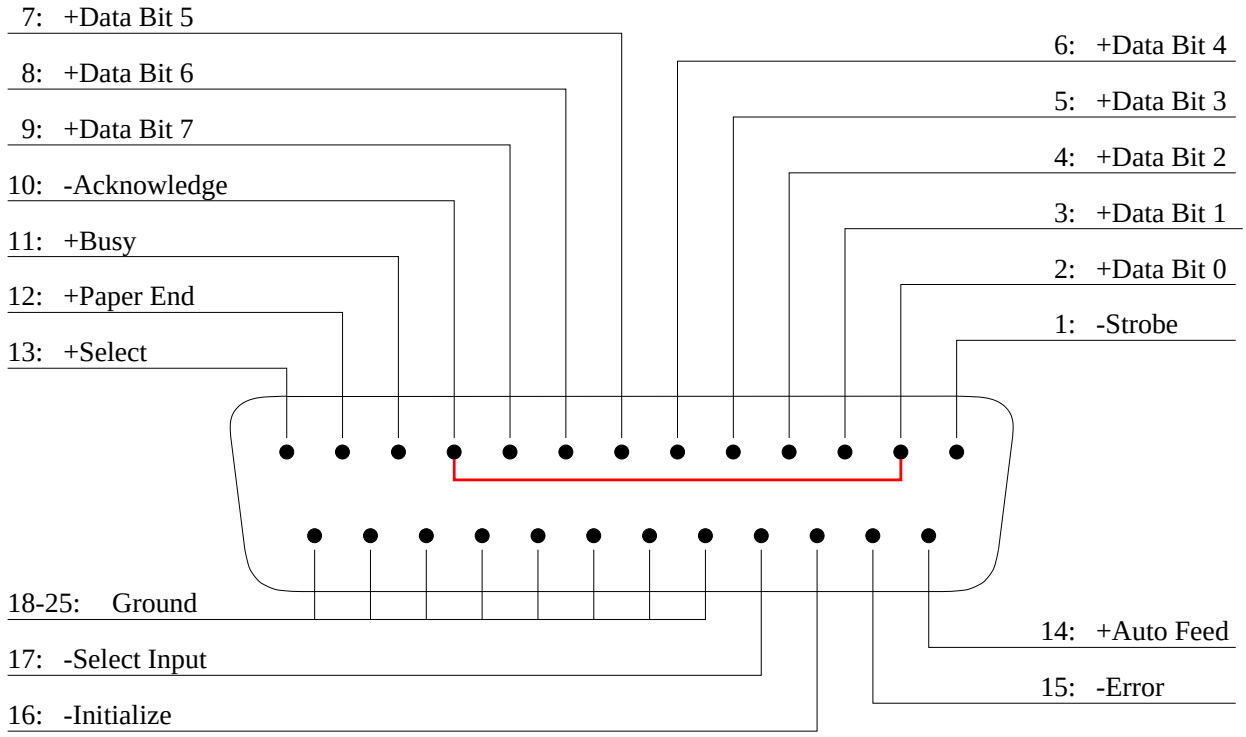



**Abbildung 10.2:** Die Datenleitung 0 und die Acknowlege-Leitung sind zur Messung der Interrupt Latency kurzgeschlossen.

Dies hat zur Folge, dass der Computer am Parallelport einen Interrupt auslöst, sobald er eine Eins auf die Datenleitung 0 legt. In den folgenden Abschnitten wird ein Treiber für Linux und LynxOS vorgestellt, der die *Interrupt Latency* bzw. *Interrupt Dispatch Latency* misst. Der Treiber legt eine Eins auf die Datenleitung und löst hiermit einen Interrupt aus. Er misst die Zeit zwischen dem Herausschreiben und dem Eintritt in den Interrupt-Handler und erhält somit die *Interrupt Latency*. Auf ähnliche Weise misst er die *Interrupt Dispatch Latency*. Die Treiber wurden so entwickelt, dass sie zur Laufzeit nachgeladen werden. Bei Linux und LynxOS können aber auch Treiber entwickelt werden, die zum Kern statisch hinzugebunden werden. Im übrigen stellt der Parallelport ein zeichenorientiertes Gerät dar.

**ACHTUNG**: Da die verwendeten Programme und Treiber nur dann realistische Zeiten messen, wenn sie die einzigen ausgeführten Prozesse sind, dürfen keine anderen Treiber auf die parallele Schnitstelle zugreifen. Dies kann u.U. auch zu Systemabstürzen

führen und ist deshalb vor Beginn des Tests sicherzustellen.

### 10.3.1 Der Linux-Treiber zur Interrupt-Messung

Unter Linux können dynamische Treiber durch den Befehl

```
insmod Name_des_Treibers
```

nachgeladen und mit

```
rmmod Name_des_Treibers
```

wieder aus dem System entfernt werden. Beim Laden des Treibers wird stets die Funktion *init_module* aufgerufen, die den Treiber initialisiert. Wird der Treiber aus dem System entfernt, so wird die Funktion *cleanup_module* aufgerufen. Für die Messung der Interruptlatenzzeiten werden nur drei Funktionen benötigt. Somit sind nicht alle Einstiegspunkte des Treibers definiert. Beim Compilieren eines Treibers unter Linux sollten die Makros *MODULE* und *__KERNEL__* gesetzt sein. Es ist zu beachten, dass in einem Treiber keine Funktionen aus der C-Library verwendet werden. Somit funktioniert der *printf*-Befehl nicht. Im Kern existiert aber ein ähnlicher Befehl. Unter Linux kann z.B. im Treiber die Funktion *printf* durch die Funktion *printk* ersetzt werden.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/semaphore.h>

#include <lpt.h>

#define LPT_MAJOR     75
#define LPT_DEVNAME   "/dev/lpt"

long long t1 = 0;
long long t2 = 0;
long long t3 = 0;
volatile int done = 0;
int use = 0;
struct semaphore  use_lock = MUTEX;

extern struct file_operations lptfops;

__inline__ unsigned long long read_timestamp(void)
{
     unsigned long long x = 0;
     asm volatile ("rdtsc" : "=A" (x));
     return x;
}
```

```
/* interrupt handler */
void irq_handler(int irq, void* id, struct pt_regs* regs)
{
    /* read time stamp */
    t2 = read_timestamp();

    /* clear parallel port */
    outb(0x00, LPT_PORT);

    done = 1;

    /* read time stamp */
    t3 = read_timestamp();
}

/* Install Driver */

int init_module(void)
{
    int stat = 0;

    /* initialize printer */
    outb(0x00, LPT_PORT);
    outb(0x00, LPT_PORT);

    stat = register_chrdev(LPT_MAJOR, LPT_DEVNAME, &lptfops );
    if (stat < 0) {
        printk("lpt failed to register\n");
        return(1);
    }

    request_irq(LPT_IRQ, irq_handler, SA_INTERRUPT, "lpt", NULL);

    outb(0x10, LPT_PORT+2);

    return 0;
}

/* Uninstall Driver */
void cleanup_module(void)
{
    outb(0x00, LPT_PORT+2);

    free_irq(LPT_IRQ, NULL);
```

```
    unregister_chrdev(LPT_MAJOR, LPT_DEVNAME);
}

int lptopen(struct inode* _inode, struct file *f)
{
    int ret = 0;

    /* enter critical section */
    down(&use_lock);

    /* only one process can open this device */
    if (use == 0)
        use = 1;
    else
        ret = EBUSY;

    /* leave critical section */
    up(&use_lock);

    return ret;
}

int lptclose(struct inode* _indoe, struct file* f)
{
    /* enter critical section */
    down(&use_lock);

    use = 0;

    /* leave critical section */
    up(&use_lock);

    return 0;
}

int lptioctl(struct inode* _inode, struct file* f,
             unsigned int command, unsigned long arg)
{
    double dummy = 0.0;

    switch(command) {
    case GETIRQLATENCY:
        t1 = read_timestamp();
        outb(0x01, LPT_PORT);
        while(done == 0)
```

```
            ;
        done = 0;
        dummy = (double) (t2 - t1);
        copy_to_user((char*) arg, (char*) &dummy, sizeof(dummy));
        break;
    case GETDISPATCHTIME:
        outb(0x01, LPT_PORT);
        while(done == 0)
            ;
        t1 = read_timestamp();
        done = 0;
        dummy =  (double) (t1 - t3);
        copy_to_user((char*) arg, (char*) &dummy, sizeof(dummy));
        break;
    default:
        return -EINVAL; /* invalid argument */
    }

    return 0;
}

ssize_t lptwrite(struct file* f, const char* buff, size_t length,
                 loff_t* ppos)
{
    char c;

    if (length != 1)
        return -EINVAL;

    copy_from_user(&c, buff, 1);
    outb(c, LPT_PORT);

    return 1;
}

ssize_t lptread(struct file* f, char* buff, size_t length,
                loff_t* ppos)
{
    return -ENXIO;
}

struct file_operations lptfops = {
    NULL,        /* lseek               */
    lptread,     /* read                */
    lptwrite,    /* write               */
    NULL,        /* readdir             */
```

```
    NULL,       /* poll                 */
    lptioctl,   /* ioctl                */
    NULL,       /* mmap                 */
    lptopen,    /* open                 */
    NULL,       /* flush                */
    lptclose,   /* release              */
    NULL,       /* fsync                */
    NULL,       /* fasync               */
    NULL,       /* check_media_change   */
    NULL,       /* revalidate           */
    NULL        /* loc                  */
};
```

**Abbildung 10.3:** Treiber zur Messung der Interruptlatenzzeiten unter Linux

In der Initialisierungs-Routine *init_module* teilt der Treiber mit dem Befehl *register_chdrv* dem Betriebssystem mit, welches zeichenorientierte Gerät er verwaltet. Hierfür benötigt er den genauen Pfad des Gerätes (hier */dev/lpt*) sowie die Major-Number (hier 75). Außerdem ordnet der Treiber durch die Funktion *request_irq* dem Interrupt des Parallelports den Handler *irq_handler* zu. Außerdem wird der Parallelport initialisiert. Hierzu wird die Funktion *outb* verwendet. Mit dieser Funktion kann an einer bestimmten IO-Adresse ein Wert hinein geschrieben werden. Der Parallelport liegt in diesem Beispiel an der IO-Adresse 0x378 und verwendet den Interrupt 7. In der Funktion *cleanup_module* werden die angeforderten Ressourcen wieder freigegeben. Hierzu werden die Funktionen *free_irq* sowie *unregister_chrdev* verwendet. Um das Gerät ansprechen zu können, muss man im Anwendungsprogramm das Gerät durch den Befehl *open* öffnen. Der *open*-Befehl ruft die Funktion *lptopen* des Treibers auf. Da nur ein Programm das Gerät verwenden darf, muss in dieser Funktion festgestellt werden, ob das Gerät bereits geöffnet wurde. Hierzu wird die Variable *use* benutzt, die auf Eins gesetzt wird, wenn das Gerät verwendet wird. Die Funktion *lptopen* schlägt fehl, falls die Variable *use* bereits auf Eins gesetzt ist. Der Zugriff auf die Variable *use* wird durch eine Semaphore geschützt, damit stets exklusiv auf die Variable zugegriffen wird. Wird das Anwendungsprogramm beendet, so gibt es durch den *close*-Befehl die Ressourcen wieder frei. Der *close*-Befehl ruft die Funktion *lptclose* des Treibers auf, der die Variable *use* wieder auf Null setzt, damit ein anderes Programm den Gerätetreiber verwenden darf.

Die Hauptfunktion des Treibers stellt die Funktion *lptioctl* dar. Diese Funktion wird durch die *ioctl*-Funktion aufgerufen. Die Variable *request* beschreibt, welcher Befehl ausgeführt werden soll. Parameter oder Rückgabewerte werden in *arg* abgelegt. Da *ioctl* einen char* erwartet, ist ein hier evtl. ein typecast erforderlich. Hier beschreibt *command*, welche Zeit gemessen werden soll. Wird ein Wert übergeben, der dem Makro *GETIRQLATENCY* entspricht, so wird die *Interrupt Latency* gemessen Entspricht *command* dem Wert für das Makro *GETDISPATCHTIME,* so wird die *Interrupt Dispatch Latency* gemessen. Die Rückgabe des gemessenen Werts (Typ *double*) erfolgt in *arg*. Die beiden Makros *GETIRQLATENCY* und *GETDISPATCHTIME* sind im Header *lpt.h*, den man auf der Homepage des Praktikums findet, definiert. Im Kern können keine Funktionen aus der C-Library verwendet werden. Um die Interruptlatenzzeiten zu messen, kann aber der P*erformance Counter* des Pentiums ausgewertet werden. Dieser Zähler wird bei jedem Taktzyklus um Eins erhöht. Mit der Funktion *read_timestamp* kann der Zähler ausgele-

sen werden. Die Funktion gibt eine Variable vom Typ *long long*, die eine 64 Bit lange Integer-Zahl darstellt, zurück. Angenommen, es soll die *Interrupt Latency* gemessen werden, so liest der Treiber den Zähler vor dem Auslösen des Interrupts und als erstes im Interrupt-Handler aus. Die Differenz dieser beiden Zähler stellt die Anzahl der Taktzyklen dar, die zwischen dem Auslösen des Interrupts und dem Eintritt in den Interrupt-Handler vergeht. Die Differenz soll dem Programm als Ergebnis übergeben werden. Unter Linux kann der Kern nicht direkt auf den Speicher eines Anwendungsprogrammes zugreifen, da Linux eine virtuelle Speicherverwaltung verwendet und somit die Adressräume nicht identisch sind. Im Kern kann aber eine virtuelle Adresse umgewandelt werden, so dass man die Daten in den Adressraum des Anwendungsprogrammes kopieren kann. Dieser Kopiervorgang kann am einfachsten durch die Funktion *copy_to_user* durchgeführt werden, die die Daten vom Adressraum des Kerns in den Adressraum eines Programmes kopiert.

Der Interrupt-Handler *irq_handler* ermittelt beim Eintritt und Verlassen den Wert des *Performance Counters*, damit man in der Funktion *lptioctl* die *Interrupt Latency* bzw. *Interrupt Dispatch Latency* berechnen kann. Zwischendurch wird der Parallelport auf den Anfangszustand zurückgesetzt und signalisiert durch das Setzen der Variablen *done*, dass der Interrupt-Handler den Interrupt bearbeitet hat.

Neben der Messung der *Interrupt Latency* bzw. *Interrupt Dispatch Latency* kann durch den Treiber ein Zeichen auf dem Parallelport gelegt werden. Dies wird durch die Funktion *lpt_write* realisiert. Der Anwender kann somit über die Funktion *write* (siehe Manpage) ein einzelnes Zeichen über den Parallelport hinaus schreiben. Diese Eigenschaft wird in einem späteren Versuch benötigt, um ein externes Gerät direkt anzusteuern. Für die Messung der *Interrupt Latency* bzw. *Interrupt Dispatch Latency* ist diese Funktion nicht weiter interessant. Mit Hilfe der Variablen *lptfops* vom Typ *file_operations* wird definiert, welche Funktion des Treibers welchem Einstiegspunkt entspricht.

### 10.3.2 Der LynxOS-Treiber zur Interrupt-Messung

In diesem Abschnitt wird der LynxOS-Treiber vorgestellt, der dem Linux-Treiber aus der Abbildung 10.3 entspricht. Es wird sich zeigen, dass der Unterschied zwischen den beiden Treibern gar nicht so groß ist. Einige Befehle ähneln den Befehlen unter Linux. Zum Beispiel wird eine Ausgabe unter LynxOS durch *cprintf* erzeugt, während Linux eine Ausgabe durch *printk* erzeugt. Im Abschnitt 10.3.1 wurde der Linux-Befehl *outb* erläutert. Unter LynxOS existiert im Prinzip der gleiche Befehl. Hier heißt er aber *_outb* und die beiden Argumente sind gegenüber Linux vertauscht.

```
#include <conf.h>
#include <kernel.h>
#include <mem.h>
#include <sys/file.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <dldd.h>
#include <port_ops_x86.h>

#include <lpt.h>
```

```
extern int iointset    _AP((int vector,
                       void (*function)(void *), char *argument));
extern int iointclr    _AP((int));
extern int sysfree     _AP((char *, int));
extern void cprintf    _AP((char *, ...));
extern int stgetprio   _AP((int stid));
extern int stgetid     _AP((void));
extern void stsetprio  _AP((int stid, int prio));

__inline__ unsigned long long read_timestamp(void)
{
    unsigned long long x = 0;
    asm volatile ("rdtsc" : "=A" (x));
    return x;
}

struct lptstatics {
    int port;            /* port address                */
    int irq;             /* irq number                  */
    int use;             /* use any procecc this device  */
};


long long t1 = 0;
long long t2 = 0;
long long t3 = 0;
volatile int done = 0;

/* interrupt handler */
void irq_handler(void* arg)
{
    /* read time stamp */
    t2 = read_timestamp();

    /* clear parallel port */
    __outb(((struct lptstatics*) arg)->port, 0x00);

    done = 1;

    /* read time stamp */
    t3 = read_timestamp();
}

/* Install Driver */
char* lptinstall(struct lptinfo* info)
```

```
{
    struct lptstatics *s;

    __outb(info->port, 0x00);
    __outb(info->port+2, 0x00);

    s = (struct lptstatics *) sysbrk(sizeof(struct lptstatics));
    if (!s) {
        cprintf("LPT: Could not allocate memory\n");
        return (char*) SYSERR; /* could not allocate memory */
    }

    /* initialize statics */
    s->port = info->port;
    s->irq = info->irq;
    s->use = 0;

    /* initialize lpt device */
    iointset(32+s->irq, irq_handler, (char*) s);
    __outb(s->port+2, 0x10);

    return ((char *) s); 4
}

/* Uninstall Driver */
int lptuninstall(struct lptstatics* s)
{
    __outb(s->port+2, 0x00);
    iointclr(32+s->irq);
    sysfree((char*) s, sizeof(struct lptstatics));

    return OK;
}

int lptopen(struct lptstatics* s, int devno, struct file *f)
{
    int ps, ret = OK;

    /* enter critical section */
    sdisable(ps);

    /* only one process can open this device */
    if (s->use == 0) {
        s->use = 1;

        /* leave critical section */
```

```
        srestore(ps);
    } else {
        /* leave critical section */
        srestore(ps);
        ret = SYSERR;
        pseterr(EBUSY);
    }

    return ret;
}

int lptclose(struct lptstatics* s, struct file* f)
{
    int ps;

    /* enter critical section */
    sdisable(ps);

    s->use = 0;

    /* leave critical section */
    srestore(ps);


    return OK;
}

int lptioctl(struct lptstatics* s, struct file* f, int command,
             char* arg)
{
    /* set kernel thread priority */
    stsetprio(stgetid(), (stgetprio(stgetid()) << 1)+1);

    switch(command) {
    case GETIRQLATENCY:
        t1 = read_timestamp();
        __outb(s->port, 0x01);
        while(done == 0)
            ;
        done = 0;
        *((double*) arg) = (double) (t2 - t1);
        break;
    case GETDISPATCHTIME:
        __outb(s->port, 0x01);
        while(done == 0)
            ;
```

```
        t1 = read_timestamp();
        done = 0;
        *((double*) arg) = (double) (t1 - t3);
        break;
    default:
        pseterr(EINVAL); /* invalid argument */
        return SYSERR;
    }

    return OK;
}

int lptwrite(struct lptstatics *s, struct file* f, char* buff,
             int count)
{
    if (count != 1)
    {
        pseterr(EINVAL);
        return SYSERR;
    }

    __outb(s->port, *buff);


    return 1;
}

int lptread(struct lptstatics *s, struct file* f, char* buff,
             int count)
{
    pseterr(ENXIO);
    return SYSERR;
}

extern int ionull();

static struct dldd entry_points = {
    lptopen, lptclose, lptread, lptwrite, ionull, lptioctl,
    lptinstall, lptuninstall, (char*) 0
};
```

**Abbildung 10.4:** Treiber zur Messung der Interruptlatenzzeiten unter LynxOS

Jeder Eintrittspunkt in LynxOS bekommt einen Zeiger auf eine Datenstruktur übergeben, die zuvor vom Programmierer definiert wurde. In diesem Beispiel handelt es sich um die Datenstruktur *lptstatics*. Für diese Datenstruktur muss die Initialisierungs-Routine *lptinstall* durch die Funktion *sysbrk* Speicher anfordern und anschließend initialisieren. Wie

diese Datenstruktur initialisiert werden soll, erfährt das Programm aus der Datenstruktur *lptinfo*, die beim Installieren des Treibers gesetzt wird. In diesem Beispiel wird die IO-Adresse des Parallelports übergeben. Beim Linux-Treiber wurde die IO-Adresse durch ein Makro definiert. Dies hat zur Folge, dass bei einer Änderung der IO-Adresse des Parallelports der Linux-Treiber neu compiliert werden muss. Bei LynxOS kann die IO-Adresse zur Laufzeit definiert werden. Dies vereinfacht die Handhabung des Treibers enorm. Unter Linux existiert ebenfalls eine solche Möglichkeit, doch ist deren Erläuterung für dieses Praktikum zu komplex.

In der Initialisierungs-Routine wird wie bei Linux der Interrupt-Handler des Parallelports festgelegt. Hierzu wird die Funktion *iointset* verwendet. Auf x86-basierten Systemen muss unter LynxOS zur Nummer des Interrupts noch 32 addiert werden. Wird der Treiber aus dem System entfernt, so wird die Funktion *lptuninstall* aufgerufen. In dieser Funktion wird der allokierte Speicher durch *sysfree* und der Interrupt des Parallelports durch *iointclr* wieder freigegeben. Die Funktionen *lptopen* und *lptclose* entsprechen im Prinzip den gleichnamigen Funktionen des Linux-Treibers. Allerdings wird der Zugriff auf die Variable *use* nicht durch eine Semaphore synchronisiert. In LynxOS kann durch den Befehl *sdisable* ein Kontext-Wechsel verboten werden. Mit *srestore* wird der Verbot eines Kontex-Wechsels wieder aufgehoben. Innerhalb dieser beiden Befehle hat der Treiber exklusiven Zugriff auf die Variable *use*, da kein anderer Prozess Rechenzeit bekommen kann. Die Funktionen *lptread, lptwrite, lptioctl* und *irq_handler* entsprechen den gleichnamigen Funktionen des Linux-Treibers. Mit Hilfe der Variablen *entry_points* vom Typ *dldd* wird definiert, welche Funktion des Treibers welchem Einstiegspunkt entspricht.

## 10.4 Messung der Interrupt-Latenzzeit unter QNX

Da QNX auf einem Mikrokern basiert, sind alle Treiber nicht im Kern implementiert, sondern laufen als eigenständige Programme im System. Unter QNX können somit Programme einen eigenen Interrupt-Handler definieren. Durch den Befehl **InterruptAttach** (s.o.) wird der Interrupt-Handler *handler* beim System angemeldet, so dass beim Auslösen des Interrupts mit der Nummer *irq* der Interrupt-Handler *handler* aufgerufen wird. Die Funktion **InterruptAttach** liefert eine Identifikationsnummer für den Interrupt-Handler zurück, mit dessen Hilfe man den Interrupt-Handler wieder mit **InterruptDetach** beim System abmelden kann.

Die Abbildung 10.5 stellt ein Programm unter QNX dar, dass die *Interrupt Latency* und *Interrupt Dispatch Latency* misst. Wie im vorherigen Abschnitt werden die Zeiten durch den „Performance Counter“ des Pentium Prozessors ermittelt. Die Differenz stellt die Anzahl der vergangenen Taktzyklen dar. Um eine Größe zu erhalten, mit der sinnvoll verglichen werden kann, muss die Differenz durch den Systemtakt in MHz (hier: 200 MHz) geteilt werden. Somit erhält man aus der Anzahl der vergangenen Taktzyklen Microsekunden.

Das Programm ist relativ einfach. Es meldet seinen Interrupt-Handler beim System an und löst alle 100 ms einen Interrupt aus. Zunächst misst es tausendmal die *Interrupt Latency* und anschließend tausendmal die *Interrupt Dispatch Latency*. Die Ergebnisse werden in eine Datei geschrieben, so dass sie mittels Gnuplot (siehe Anhang B) visualisiert werden können.

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

```
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <sys/sched.h>
#include <sys/neutrino.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/file.h>

#include <hw/inout.h>
#include <sys/mman.h>
#include <inttypes.h>
#include <atomic.h>
#include <pthread.h>
#include <errno.h>


#define LPT_PORT    0x378
#define LPT_IRQ     7
#define N           100
#define TAKT        200.0
#define TIC_TIME    (double)(1.0 / ((double) TAKT))

double            t[N];
int               id = -1;
FILE*             file = NULL;
volatile uint64_t t1 = 0;
volatile uint64_t t2 = 0;
volatile uint64_t t3 = 0;
volatile uint64_t t4 = 0;
volatile int      done = 0;
volatile uint64_t x = 0;
uintptr_t         port;

timer_t           tid;
struct sigevent   event;

const char*       irq_latency_file = "irq_qnx.dat";
const char*       dispatch_time_file = "dispatch_qnx.dat";

/* read timestamp */
__inline__ uint64_t
read_timestamp(void)
{
  asm volatile ("rdtsc" : "=A" (x));
```

```
  return x;
}

/* interrupt handler */
const struct sigevent*
irq_handler(void *area, int id)
{
  /* read time stamp */
  t2 = read_timestamp();
  /* Mask Interrupt */
  InterruptMask(LPT_IRQ, id);
  /* clear parallel port */
  out8(port, 0x00);
  atomic_set(&done, 0x01);
  /* read time stamp */
  t3 = read_timestamp();
  return ( &event );
}

void
ExitHandler(int sig)
{
  fprintf(stderr, "Got SIGINT or SIGTERM\n");
  if (file != NULL)
    fclose(file);
  if (tid!= -1)
    timer_delete(tid);
  if (id!=-1)
    InterruptDetach(id);
  exit(0);
}


int
main(int argc, char* argv[])
{
  int                 i;
  double              latency=0.0,dispatch=0.0,min,max,average;
  struct sched_param          param;
  struct sigaction    sINT, sTERM;
  struct timespec     rqtp = {0, 200000000};

  fprintf(stderr, "\nStarte Benchmark...\n\n");
  event.sigev_notify = SIGEV_INTR;
```

```
  /* obtain I/O privileges */
  if((ThreadCtl (_NTO_TCTL_IO, NULL))==-1) {
    perror("ThreadCtl");
  }

  /* map port */
  if (!(port=mmap_device_io(3,LPT_PORT))) {
    perror("mmap_device_io");
    exit(1);
  }

  /* attach irq */
  if (!(id = InterruptAttach(LPT_IRQ, irq_handler, NULL,0,0))) {
    perror("InterruptAttach");
    exit(1);
  }

  /* define handler for SIGINT */
  sINT.sa_handler = ExitHandler;
  sINT.sa_flags = 0;
  if (sigaction(SIGINT, &sINT, NULL) < 0) {
    perror("sigaction SIGINT");
    exit(1);
  }

  /* define handler for SIGTERM */
  sTERM.sa_handler = ExitHandler;
  sTERM.sa_flags = 0;
  if (sigaction(SIGTERM, &sTERM, NULL) < 0) {
    perror("sigaction SIGTERM");
    exit(1);
  }

  /* Clear parallel port */
  out8(port, 0x00);
  out8(port+2, 0x10);

  /* POSIX compatible scheduling: get maximum priority */
  param.sched_priority = sched_get_priority_max(SCHED_FIFO);
  fprintf(stderr, "max. priority (fifo) = %i\n",
param.sched_priority);
  if (sched_setscheduler(0, SCHED_FIFO, &param) == -1)
    perror("sched_setscheduler");
  if (sched_getparam(0, &param) == -1)
    perror("sched_getparam");
```

```
fprintf(stderr, "priority = %i\n", param.sched_priority);

/* do the tests */
fprintf(stderr, "measuring latency \n");
file = fopen(irq_latency_file, "w");
done =0;
/* GETIRQLATENCY: */
for (min=100000.0, max=average=0.0, i=0; i<N; i++) {
  /* sleep a while */
  nanosleep(&rqtp, NULL);
  /* read first timestamp */
  t1 = read_timestamp();
  /* set pin 2 to high */
  out8(port, 0x01);
  /* wait for isr to set done */
  while (!done)
    ;
  /* read fourth timestamp */
  t4 = read_timestamp();
  /* calculate latency */
  latency = (double) t2 - (double) t1;
  latency*=TIC_TIME;
  (min>latency)&&(min=latency);
  (max<latency)&&(max=latency);
  average+=latency;
  /* print out some data samples*/
  if (i%(N/10) == 0) {
    fprintf(stderr, "irq latency = %f usec\n", latency);
  }
  fprintf(file, "%f %f\n", (double) i, latency);
  /* arm for the next shot */
  atomic_set(&done,0x00);
  InterruptUnmask(LPT_IRQ,id);
}
fclose(file);
average/=(double)N;
/* write out results */
fprintf(stderr, "average irq latency = %f usec\n", average);
fprintf(stderr, "min = %f usec, max =%f usec\n",min,max);
/* GETDISPATCHLATENCY */
fprintf(stderr, "now measuring dispatch time \n");
file = fopen(dispatch_time_file, "w");
done=0;
for (min=100000.0,max=average=0.0, i=0; i<N; i++) {
  /* wait 0,2 secs */
```

```
    nanosleep(&rqtp, NULL);
    /* read first timestamp */
    t1 = read_timestamp();
    /* set pin 2 to high */
    out8(port, 0x01);
    /*wait for isr to set done */
    while (!done)
      ;
    /* read fourth timestamp */
    t4 = read_timestamp();
    /* calculate latency */
    dispatch = (double) t4 - (double) t3;
    dispatch*=TIC_TIME;
    (min>dispatch)&&(min=dispatch);
    (max<dispatch)&&(max=dispatch);
    average+=dispatch;
    /* output some data samples */
    if (i%(N/10) == 0) {
      fprintf(stderr, "dispatch time = %f usec\n", dispatch);
    }
    fprintf(file, "%f %f\n", (double) i, dispatch);
    /* arm for the next shot */
    atomic_set(&done,0x00);
    InterruptUnmask(LPT_IRQ,id);
  }
  /* close file */
  fclose(file);
  average/=(double)N;
  /* write out results */
  fprintf(stderr, "average dispatch time = %f usec\n", average);
  fprintf(stderr, "min = %f usec, max =%f usec\n",min,max);
  /* set port to all zero */
  out8(port, 0x00);
  out8(port+2, 0x10);
  /* give back the interrupt */
  InterruptDetach(id);
  fprintf(stderr, "\nBenchmark wird beendet\n\n");
  return 0;
}
```

**Abbildung 10.5:** Messung der Interruptlatenzzeiten unter QNX 6

# 11Praktikumsaufgaben

## 11.1 Versuch 1:Leistungsuntersuchungen in Realzeitbetriebssystemen

Es existieren eine Reihe von Kennwerte, die man bei der Leistungsuntersuchung eines Realzeitbetriebssystems beachten muss. Um den Rahmen dieses Praktikums nicht zu sprengen, wird in diesem Versuchsblock nur die *Interrupt Latency* und *Interrupt Dispatch Latency* betrachtet. In Abschnitt 10 wurde ein Programm für QNX vorgestellt, dass die *Interrupt Latency* und die *Interrupt Dispatch Latency* misst. Das Programm erzeugt die Dateien *irq.dat* und *dispatch.dat*. Die Datei *irq.dat* enthält die gemessenen Zeiten für die *Interrupt Latency*, während die Datei *dispatch.dat* die gemessenen Zeiten für die *Dispatch Latency* enthält. Das Programm muss nicht neu erzeugt werden, da sich bereits eine compilierte Version des Programmes auf der Homepage des Praktikums befindet.

**1. Aufgabe:** *Führen Sie das Programm unter zwei verschiedenen Testbedingungen aus. Beim ersten Durchlauf sollte auf dem System kein weiteres Programm laufen. Im zweiten Durchlauf sollte das System die Messung unter hoher Last durchführen. Schreiben Sie hierzu ein Programm, dass mit normaler Priorität ständig lesend und schreibend auf die lokale Festplatte zugreift und somit Interrupts auslöst. Die Datei, die das Programm hierzu erzeugt, sollte mindestens 20 MByte groß werden. Die Ergebnisse sollten anschließend durch Gnuplot (siehe Abschnitt B) visualisiert werden.*

In Abschnitt 10 wurde ein Treiber für Linux und LynxOS vorgestellt, mit dessen Hilfe die *Interrupt Latency* sowie die *Interrupt Dispatch Latency* gemessen werden kann. Dieser Treiber kann über die Datei */dev/lpt* mit den Befehlen *open, close* bzw. *ioctl* angesprochen werden.

**2. Aufgabe:** *Entwickeln Sie ein Programm, das die Interrupt Latency sowie die Interrupt Dispatch Latency mit Hilfe des Treibers aus Abschnitt 10 unter Linux sowie LynxOS misst. Das Programm sollte die höchste Priorität im System verwenden und eintausend mal die Interupt Latency sowie die Interrupt Dispatch Latency messen. Zwischen jeder Messung sollte das Programm 100 ms warten, damit auch andere Programme im System Rechenzeit bekommen. Nach jedem Iterationsschritt sollten die Ergebnisse in eine Datei geschrieben werden. Die Ergebnisse sollten anschließend durch Gnuplot (siehe Anhang B:.) visualisiert werden. Führen Sie die Messung unter zwei verschiedenen Testbedingungen durch. Beim ersten Durchlauf sollte auf dem System kein weiteres Programm laufen. Im zweiten Durchlauf sollte das System die Messung unter hoher Last durchführen. Schreiben Sie hierzu ein Programm, dass mit normaler Priorität ständig lesend und schreibend auf die lokale Festplatte zugreift und somit Interrupts auslöst. Die Datei, die das Programm hierzu erzeugt, sollte mindestens 20 MByte groß werden.*

Nachdem die Betriebssysteme LynxOS, QNX und Linux näher betrachtet wurden, wird im nächsten Versuch RT Linux genauer untersucht.

**3.Aufgabe:** *Entwickeln Sie ein Kernelmodul(Treiber), der die Interrupt Latency sowie die Interrupt Dispatch Latency unter RT Linux misst. Das Modul soll einen Thread erzeugen, der eintausend mal die Interrupt Latency sowie die Interrupt Dispatch Latency misst. Zwischen jeder Messung sollte der Thread 100 ms warten, damit auch andere Programme im System Rechenzeit bekommen. Dazu bietet es sich an, diesen Thread mit 100ms periodisch zu machen. Nach jedem Iterationsschritt sollten die Ergebnisse durch eine Real-Time FIFO zu einem normalen Linux-Programm gesendet werden, das die Ergebnisse in eine Datei schreibt. Die Ergebnisse sollten anschließend durch Gnuplot (siehe Anhang B:.) visualisiert werden. Führen Sie die Messung unter zwei verschiedenen Testbedingungen durch. Beim ersten Durchlauf sollte auf dem System kein weiteres Programm laufen. Im zweiten Durchlauf sollte das System die Messung unter hoher Last durchführen. Schreiben Sie hierzu ein „normales“ Linux-Programm, dass ständig lesend und schreibend auf die lokale Festplatte zugreift und somit Interrupts auslöst. Die Datei, die das Programm hierzu erzeugt, sollte mindestens 20 MByte groß werden. Vergleichen Sie nun QNX, LynxOS, Linux und RT Linux miteinander. Worin liegen die Stärken der einzelnen Betriebssysteme?*

Wie bereits erwähnt wurde, basiert QNX auf einem Mikro-Kern, während Linux und LynxOS klassische monolithische Betriebssysteme darstellen.

**4.Aufgabe:** *Stellen Sie eine Tabelle auf, in der Sie die beiden Betriebssystemarchitekturen (Mikro-Kern, monolytisches Betriebssystem) miteinander vergleichen. Berücksichtigen Sie insbesondere die Ergebnisse aus diesem Versuchsblock.*

## 11.2 2. Versuch: Servoansteuerung mit Real-Time Linux

Im nächsten Versuch soll ein Realzeit-Programm unter RT Linux entwickelt werden, das Servomotoren über den Parallelport ansteuert. Handelsübliche Servomotoren für den Modellbau haben einen nominalen Wirkbereich von ca. 90 Grad, Stellkräfte von 30-159 Nm und drei elektrische Anschlussleitungen: Masse, Betriebsspannung und Steuersignal. Die Betriebsspannung darf zwischen 4,5 und 6 Volt liegen. Alle 10 bis 30 Millisekunden muss ein TTL-kompatibler (High-) Impuls von 1 bis 2 Millisekunden Länge auf die Signalleitung gelegt werden. Die Länge des Impulses bestimmt die Stellung des Servos (z.B.: 1ms=Links, 1,5ms=Mittelstellung, 2ms=Rechts, vgl. dazu Abbildung 11.1).

Jeder billige NE555-Taktgeberbaustein für 25 Cent kann das gewünschte Signal erzeugen - aber ein gewöhnlicher PC wäre mit dieser Aufgabe hoffnungslos überfordert, da die Länge des Signal-Impulses sehr exakt eingehalten werden muss! Ansonsten zittert der Servo sehr stark oder geht gar für mehrere 100 Millisekunden in den Schlafmodus, weil das Steuersignal ausgeblieben ist. 10 Mikrosekunden Abweichung bei der Signalerzeugung bedeutet immerhin etwa 1 Grad Abweichung bei der Servostellung. Dies sind sehr hohe Zeitanforderungen für ein PC-Betriebssystem! Aber unter RT Linux lassen sich sogar mehrere dieser heiklen Motoren ansteuern. Dabei erzeugt ein RT-Task fortlaufend nacheinander die Impulse für die Servos.

Die Servomotoren werden über den Parallelport an den PC angeschlossen. Die Praktikumsteilnehmer sollen ein Realzeit-Programm unter RT Linux entwerfen, das die Servo-

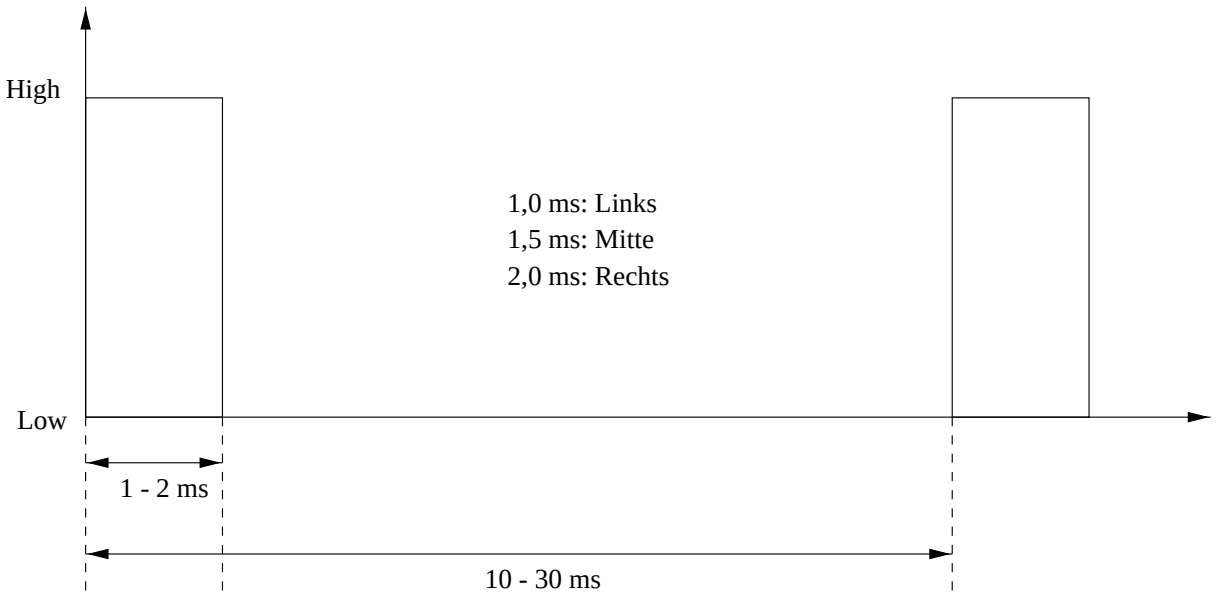



**Abbildung 11.1:** Timing-Diagramm zur Servoansteuerung

motoren ansteuert. Das zu entwickelnde Programm soll über eine Fifo mit dem vom Lehrstuhl zur Verfügung gestellten Programm *servo_rtlinux* kommunizieren. Das Programm *servo_rtlinux* stellt eine graphische Benutzeroberfläche dar, mit dessen Hilfe die Servos einzeln angesteuert werden können. Das Programm *servo_rtlinux* liegt im Verzeichnis http://www.lfbs.rwth-aachen.de/~tom/Praktikum/#Losungshinweise. Dabei teilt *servo_rtlinux* dem Realzeit-Programm jede Veränderung der Servoeinstellung mit. Hierzu sendet es zwei Bytes über die Fifo zum RT-Programm. Das erste Byte gibt die Nummer des Servos an, während das zweite Byte Werte zwischen 0 und 255 enthält. Hierbei steht die 0 für eine Signallänge von 1ms, während die 255 die Signallänge von 2ms darstellt. Die Zwischenwerte werden linear interpoliert. Das zu entwickelnde Realzeit-Programm soll fünf Servos ansteuern, indem es für jedes Servo einen periodischen Rechteckimpuls auf dem Parallelport erzeugt. Anhand der Breite des Impulses wird die Stellung des Servos bestimmt. Die Ausgabe auf dem Parallelport erfolgt durch die Funktion *outb*. Die Funktion ist wie folgt definiert:

```
void outb(unsigned char value, int port);
```

Für die Variable *port* wird die Portnummer des Parallelports angegeben. In diesem Fall wird der Port 0x378 verwendet. Durch die Variable *value* wird der Wert angegeben, der auf dem Parallelport ausgegeben wird. Jedes Servo kann durch einen bestimmten Wert angesprochen werden. Tabelle 11.1 kann entnommen werden, welcher Wert durch *outb* auf dem Parallelport als High-Impuls für welches Servo ausgegeben werden soll. Durch die Ausgabe einer Null auf dem Parallelport wird der High-Impuls wieder auf „Low“ zurückgesetzt.

| Servo-Nr. | Ausgabe auf dem Parallelport |
|---|---|
| 0 | 0x01 |
| 1 | 0x02 |
| 2 | 0x04 |

| Servo-Nr. | Ausgabe auf dem Parallelport |
|---|---|
| 3 | 0x08 |
| 4 | 0x10 |

**Tabelle 11.1:** Tabelle der Werte, die *outb* für das Erzeugen des Rechteckimpulses verwenden soll

**1. Aufgabe:** *Entwickeln Sie ein Realzeit-Programm für RT Linux, das fünf Servos ansteuert. Kommunizieren Sie mit dem Programm servo_rtlinux über die Fifo (von RT-Linux) mit der Nummer Eins.*

Es wäre nun interessant, einen Vergleich zwischen einem Realzeit-Programm und einem „normalen“ Programm zur Steuerung von Servos zu ziehen. Für dieses Programm wird ein sehr genauer Zeitgeber benötigt. Nun stellt sich die Frage, ob der Zeitgeber unter Linux überhaupt so genau ist.

**2. Aufgabe:** *In der Abbildung 9.2 wurde ein periodischer Timer vorgestellt. Wandeln Sie das Programm ab, so dass der Timer nach einer Millisekunde abläuft. Ermitteln Sie mit Hilfe der Funktion read_timestamp aus der Abbildung 10.3 die Zeit, die in Wirklichkeit vergangen ist. Reicht die Genauigkeit des Timers unter Linux aus, um das Problem zu lösen?*

Es liegt auf der Hand, dass der Timer unter Linux nicht genau genug ist. Die Zeitmessung in einem Betriebssystem wird in einem PC meistens durch einen Timer-Interrupt realisiert, der periodisch erzeugt wird und die Systemzeit aktualisiert. Um eine sehr genaue Auflösung der Systemzeit zu erreichen, müsste die Frequenz des periodischen Timer-Interrupts sehr hoch sein. Dies hätte den Nachteil, dass das System sehr oft in den Interrupt-Handler springen muss. Wird vom Anwender keine genaue Zeitmessung verlangt, würde hierdurch sehr viel Rechenzeit verschwendet. Aus diesem Grund wird meistens ein Wert für die Frequenz des periodischen Timers genommen, der für die meisten Anwendungen ausreicht. Leider reicht die Auflösung für die Steuerung eines Servomotors nicht aus. Ein PC besitzt aber eine Echtzeituhr, die so programmiert werden kann, dass sie mit einer Frequenz von 8192 Hz einen Interrupt auslöst. Dies stellt immerhin eine Auflösung von 122 µs dar. Mit Hilfe des Devices */dev/rtc* kann unter Linux mit der Echtzeituhr kommuniziert werden. Anhand der Abbildung 11.2 soll nun die Funktionsweise der Echtzeituhr erläutert werden.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <linux/mc146818rtc.h>

int main(int argc,char **argv)
{
```

```
    unsigned long  rtc_time;
    int            i, fd;

    fprintf(stderr, "Öffne das Device /dev/rtc\n");
    fd = open("/dev/rtc", O_RDONLY);
    if (fd == -1)
    {
        perror("open");
        exit(1);
    }

    fprintf(stderr, "Initialisiere die Rate des periodischen
                     Timers\n");
    if (ioctl(fd, RTC_IRQP_SET, 8192) != 0) {
        perror("ioctl");
        exit(1);
    }

    fprintf(stderr, "Starte periodischen Timer\n");
    if (ioctl(fd, RTC_PIE_ON, 0) != 0) {
        perror("ioctl");
        exit(1);
  }

    for(i=0; i<100000; i++) {
        read(fd, &rtc_time, sizeof(unsigned long));
        if ((i % 5000) == 0)
            fprintf(stderr, "i = %d\n", i);
    }

    fprintf(stderr, "Stoppe periodischen Timer\n");
    if (ioctl(fd, RTC_PIE_OFF, 0) != 0)
    {
        perror("ioctl");
        exit(1);
  }

    fprintf(stderr, "Schließe das Device /dev/rtc\n");
    close(fd);

    return(0);
}
```

**Abbildung 11.2:** Beispiel für die Programmierung der Echtzeituhr unter Linux

Nachdem das Device */dev/rtc* geöffnet ist, muss es zunächst durch den Befehl *ioctl(fd,*

*RTC_IRQP_SET, 8192)* initialisiert werden. Dieser Befehl legt fest, dass der Timer 8192 mal in der Sekunde eine Zahl vom Typ *unsigned long* in das Device legt. Durch den Befehl *read* kann dieser Wert vom Device ausgelesen werden. Im obigen Programm kann davon ausgegangen werden, dass zwischen zwei read-Befehlen genau 122 µs vergangen sind. Da 5000 Werte ausgelesen werden, bevor das Programm die Zahl *i* ausgibt, kann davon ausgegangen werden, dass zwischen den einzelnen Ausgaben genau 610 ms vergangen sind. Der periodische Timer wird durch den Befehl *ioctl(fd, RTC_PIE_ON, 0)* gestartet und durch den Befehl *ioctl(fd, RTC_PIE_OFF, 0)* wieder gestoppt.

Durch diesen Timer soll nun das Servo-Problem gelöst werden. Es sollte aber beachtet werden, dass ein normales Linux-Programm nicht durch den Befehl *outb* Daten auf den Parallelport legen kann. Diesen Befehl können nur Treiber verwenden. In Abbildung 10.3 wurde ein Treiber vorgestellt, mit dessen Hilfe die *Interrupt Latency* gemessen werden kann. Der Treiber kann aber auch verwendet werden, um ein Zeichen auf dem Parallelport auszugeben. Hierzu muss der Treiber nur mit dem Befehl *open("/dev/lpt", O_WRONLY)* geöffnet werden. Anschließend kann durch den Befehl *write* ein Zeichen auf dem Parallelport ausgegeben werden.

**Versuch 3:** *Entwickeln Sie ein Programm unter Linux mit Hilfe des Device /dev/rtc, das fünf Servos ansteuert. Kommunizieren Sie mit dem Programm servo_linux über die „named pipe" mit dem Namen /tmp/rtf1. Testen Sie beide Lösungen zur Steuerung von Servos. Worin liegt der Unterschied und wie lässt er sich erklären?*

## 11.3 3. Versuch: Steuerung von Schrittmotoren

In diesem Versuch soll ein Schrittmotor über den Parallelport von einem Computer angesteuert werden. Gegenüber einem „normalen" Motor werden die einzelnen Spulen des Schrittmotors direkt angesteuert. So kann jeder Schritt, den sich der Motor in einer bestimmten Richtung bewegen soll, einzeln angesteuert werden. Hierdurch ist eine viel genauere Ansteuerung möglich. Die Abbildung 11.3 stellt den Aufbau eines Schrittmotors dar.

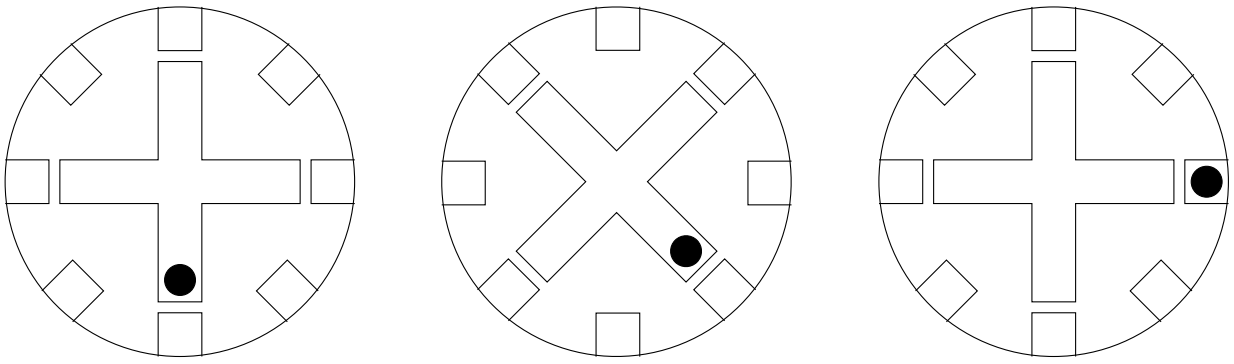



**Abbildung 11.3:** Ansteuerung eines Schrittmotors

In diesem Praktikum wird ein Schrittmotor verwendet, der nach 200 Schritten eine Umdrehung zurückgelegt hat. Angenommen, der Schrittmotor soll sich in einer

Geschwindigkeit von 2 Umdrehungen in der Sekunde drehen, so muss sich der Motor alle 2500 µs um einen Schritt bewegen. Dies kann in einem Programm relativ leicht realisiert werden, indem ein Timer verwendet wird, der alle 2500 µs ein Signal auslöst. Im Handler des Signals wird dann der Motor um einen Schritt gedreht.

Die einzelnen Schrittbewegungen könnte man einfach über den Parallelport erzeugen, indem man die Spulen über die verschiedenen Ausgangsbits des Parallelports ansteuert. Werden die bits auf dem Parallelport richtig gesetzt und damit die Spulen bestromt, so dreht sich der Motor. Wegen der geringen Ausgangsspannung des Parallelports und des hohen Stromverbrauchs des Motors wird jedoch im Praktikum eine Steuerungskarte eingesetzt. Diese bietet neben der Möglichkeit, eine stabile Spannungsquelle zu verwenden, die automatische Erzeugung des wandernden Spulenstroms. Am Parallelport muss lediglich ein Rechtecksignal ausgegeben werden.

Im letzten Versuch wurde bereits unter Linux festgestellt, dass die Auflösung, in der das Signal *SIGALRM* ausgelöst wird, zu grob für viele zeitkritische Anwendungen ist. Unter LynxOS sieht dies leider nicht anders aus.

**1. Aufgabe:** *Ermitteln Sie unter LynxOS die Auflösung des Timers mit Hilfe der Funktion clock_getres (siehe Manpage). Warum kann mit Hilfe eines normalen Timers kein Schrittmotor angesteuert werden?*

Die Zeitmessung in einem Betriebssystem wird in einem PC meistens durch einen Timer-Interrupt realisiert, der periodisch erzeugt wird und die Systemzeit aktualisiert. Um eine sehr genaue Auflösung der Systemzeit zu erreichen, müsste die Frequenz des periodischen Timer-Interrupts sehr hoch sein. Dies hätte den Nachteil, dass das System sehr oft in den Interrupt-Handler springen muss. Wird vom Anwender keine genaue Zeitmessung verlangt, würde hierdurch sehr viel Rechenzeit verschwendet. Aus diesem Grund wird meistens ein Wert für die Frequenz des periodischen Timers genommen, der für die meisten Anwendungen ausreicht. Leider reicht die Auflösung für die Steuerung eines Schrittmotors nicht aus. Ein PC besitzt aber eine Echtzeituhr, die so programmiert werden kann, dass sie mit einer Frequenz von 8192 Hz einen Interrupt auslöst. Dies stellt immerhin eine Auflösung von 122 µs dar. Durch den folgenden LynxOS-Treiber, der zur Lösung dieser Aufgabe nicht vollständig verstanden werden muss, kann die Echtzeituhr recht einfach angesteuert werden.

```
#include <conf.h>
#include <kernel.h>
#include <mem.h>
#include <errno.h>
#include <signal.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <dldd.h>
#include <port_ops_x86.h>

#include <rtc.h>
#include <drvutil.h>
```

```
#ifndef RTC_PORT
#define RTC_PORT(x)     (0x70 + (x))
#define RTC_ALWAYS_BCD  1
#endif

#define CMOS_READ(addr) ({ \
__outb(RTC_PORT(0), (addr)); \
__inb(RTC_PORT(1)); \
})
#define CMOS_WRITE(val, addr) ({ \
__outb(RTC_PORT(0), (addr)); \
__outb(RTC_PORT(1), (val)); \
})

/* control registers - Moto names */
#define RTC_REG_A               10
#define RTC_REG_B               11
#define RTC_REG_C               12
#define RTC_REG_D               13

/* register details */

#define RTC_FREQ_SELECT RTC_REG_A

#define RTC_CONTROL     RTC_REG_B
#define RTC_PIE         0x40       /* periodic interrupt enable */
#define RTC_INTR_FLAGS  RTC_REG_C

#define RTC_TIMER_OFF   0
#define RTC_TIMER_ON    1

#define FREQUENCE       8192

struct rtcstatics {
    long           interval;
    long           expires;
    int            irq;             /* irq number        */
    int            pid;             /* process id        */
    int            status;          /* timer status      */
    int            use;
};

char* rtcinstall(struct rtcinfo* info);
int rtcuninstall(struct rtcstatics* s);
int rtcopen(struct rtcstatics* s, int devno, struct file *f);
```

```
int rtcclose(struct rtcstatics* s, struct file* f);
int rtcioctl(struct rtcstatics* s, struct file* f, int command, char*
arg);
void mask_rtc_irq_bit(unsigned char bit);
void set_rtc_irq_bit(unsigned char bit);
int start_timer(struct rtcstatics* s);
int stop_timer(struct rtcstatics* s);
int set_interval(struct rtcstatics* s, unsigned long freq_ptr);

/* interrupt handler */
void rtcinterrupt(void* arg)
{
    struct rtcstatics* s = (struct rtcstatics*) arg;

    CMOS_READ(RTC_INTR_FLAGS);

    s->expires -= 1000000 / FREQUENCE;
    if (s->expires <= 0) {
        s->expires = s->interval;
        /* send SIGALRM to the process */
        _kill(s->pid, SIGALRM);
    }

}

/* Install Driver */
char* rtcinstall(struct rtcinfo* info)
{
    int                ps;
    struct rtcstatics* s;

    cprintf("Install Real-Time Clock Driver...");

    s = (struct rtcstatics *) sysbrk(sizeof(struct rtcstatics));
    if (!s) {
        cprintf("\nRTC: Not enough memory\n");
        return (char*) SYSERR;  /* could not allocate memory */
    }

    /* initialize statics */
    s->interval =0;
    s->expires = 0;
    s->irq = info->irq;
    s->pid = -1;
    s->status = RTC_TIMER_OFF;
    s->use = 0;
```

```
    iointset(32+s->irq, rtcinterrupt, (char*) s);

    /* enter critical section */
    disable(ps);
    CMOS_WRITE(((CMOS_READ(RTC_FREQ_SELECT) & 0xF0) | 0x06),
                RTC_FREQ_SELECT);
    /* leave critical section */
    restore(ps);

    cprintf("Ok\n");

    return ((char *) s);
}

/* Uninstall Driver */
int rtcuninstall(struct rtcstatics* s)
{
    cprintf("Uninstall Real-Time Clock Driver...");

    if (s->use > 0)
        cprintf("Warning: Device is busy");

    if (s->status == RTC_TIMER_ON)
        /* stop timer */
        mask_rtc_irq_bit(RTC_PIE);

    iointclr(32+s->irq);
    sysfree((char*) s, sizeof(struct rtcstatics));

    cprintf("Ok\n");

    return OK;
}

/* Open driver */
int rtcopen(struct rtcstatics* s, int devno, struct file *f)
{
    int ps;

    /* enter critical section */
    sdisable(ps);
    (s->use)++;
    /* only one process can use this driver */
    if (s->use > 1)
    {
```

```
        /* enter critical section */
        srestore(ps);
        pseterr(EAGAIN);
        return SYSERR;
    } else {
        s->pid = getpid();
        /* leave critical section */
        srestore(ps);
    }

    return OK;
}

/* Close driver */
int rtcclose(struct rtcstatics* s, struct file* f)
{
    int ps;

    /* enter critical section */
    sdisable(ps);
    (s->use)--;
    if ((s->use == 0) && (s->status == RTC_TIMER_ON))
    {
        s->status = RTC_TIMER_OFF;
        /* leave critical section */
        srestore(ps);
        /* stop timer */
        mask_rtc_irq_bit(RTC_PIE);
    } else {
        /* leave critical section */
        srestore(ps);
    }

    return OK;
}

int rtcioctl(struct rtcstatics* s, struct file* f, int command,
             char* arg)
{
    int                ret = OK;
    struct itimerspec* new_setting = (struct itimerspec*) arg;

    switch(command) {
    case SETTIMER:
        if (s->status == RTC_TIMER_ON)
```

```
            ret = stop_timer(s);

        if ((new_setting->it_interval.tv_sec != 0) ||
            (new_setting->it_interval.tv_nsec != 0))
        {
            s->expires = new_setting->it_value.tv_sec*1000000;
            s->expires += new_setting->it_value.tv_nsec/1000;

            s->interval = new_setting->it_interval.tv_sec*1000000;
            s->interval += new_setting->it_interval.tv_nsec/1000;

            ret = set_interval(s, FREQUENCE);
            if (ret != OK)
                return ret;

            ret = start_timer(s);
        }
        break;
    default:
        pseterr(EINVAL); /* invalid argument */
        ret = SYSERR;
    }

    return ret;
}

int start_timer(struct rtcstatics* s)
{
    if (s->status != RTC_TIMER_ON)
    {
        s->status = RTC_TIMER_ON;
        set_rtc_irq_bit(RTC_PIE);
    }

    cprintf("RTC: status = %d\n", s->status);

    return OK;
}

int stop_timer(struct rtcstatics* s)
{
    if (s->status == RTC_TIMER_ON)
    {
        s->status = RTC_TIMER_OFF;
        mask_rtc_irq_bit(RTC_PIE);
    }
```

```
    cprintf("RTC: status = %d\n", s->status);

    return OK;
}

int set_interval(struct rtcstatics* s, unsigned long freq)
{
    int           ps;
    unsigned long tmp = 0;
    unsigned char val;

    if ((freq < 2) || (freq > 8192))
    {
        pseterr(EINVAL); /* invalid argument */
        return SYSERR;
    }

    while(freq > (1<<tmp))
        tmp++;

    if (freq != (1<<tmp))

    {
        pseterr(EINVAL);
        return SYSERR;
    }

    /* enter critical section */
    disable(ps);
    val = CMOS_READ(RTC_FREQ_SELECT) & 0xf0;
    val |= (16 - tmp);
    CMOS_WRITE(val, RTC_FREQ_SELECT);
    /* leave critical section */
    restore(ps);

    cprintf("RTC: frequence = %d Hz\n", (int) freq);

    return OK;
}

void mask_rtc_irq_bit(unsigned char bit)
{
    int ps;
    unsigned char val;
```

```
    /* enter critical section */
    disable(ps);
    val = CMOS_READ(RTC_CONTROL);
    val &=  ~bit;
    CMOS_WRITE(val, RTC_CONTROL);
    CMOS_READ(RTC_INTR_FLAGS);
    /* enter critical section */
    restore(ps);
}

void set_rtc_irq_bit(unsigned char bit)
{
    int ps;
    unsigned char val;

    /* enter critical section */
    disable(ps);
    val = CMOS_READ(RTC_CONTROL);
    val |= bit;
    CMOS_WRITE(val, RTC_CONTROL);
    CMOS_READ(RTC_INTR_FLAGS);


    /* enter critical section */
    restore(ps);
}

extern int ionull();

static struct dldd entry_points = {
    rtcopen, rtcclose, ionull, ionull, ionull, rtcioctl,
    rtcinstall, rtcuninstall, (char*) 0
};
```

**Abbildung 11.4:** Treiber zur Ansteuerung der Echtzeituhr unter LynxOS

Dieser Treiber muss nicht in allen Details verstanden werden. Im Prinzip ähnelt der Treiber dem periodischen Timer aus Abschnitt 9.2. In einer Frequenz von 8192 Hz wird ein Interrupt ausgelöst. In dem zugehörigen Interrupt-Handler *(rtcinterrupt)* wird überprüft, ob der Timer abgelaufen ist. Ist dies der Fall, so wird das Signal *SIGALRM* an den zugehörigen Prozess über den Befehl *_kill* geschickt. Das Signal *SIGALRM* entspricht dem Signal, dass auch der normale POSIX.4 Timer erzeugt. Somit muss das Programm gegenüber der normalen Verwendung des POSIX.4 Timers kaum umgeschrieben werden. Im Eintrittspunkt *rtciotcl*, den man über die Funktion *ioctl* erreicht, wird mit Hilfe der Datenstruktur *itimerspec* (wie im Abschnitt 9.2) definiert, wann der Timer zum erstenmal abläuft und wie groß die Periode des Timers ist. Der nächste Zeitpunkt, wann der Timer abläuft, wird in der Variablen *expires* der Datenstruktur *rtcstatics* abgelegt. Diese Vari-

able wird im Interrupt-Handler um die Zeit, die zwischen zwei Interrupts vergeht, erniedrigt. Der Timer ist abgelaufen, wenn der Betrag der Variablen kleiner oder gleich Null ist. Ist dies der Fall, so wird dem Prozess ein *SIGALRM* geschickt und die Variable *expires* wird auf den nächsten Zeitpunkt gesetzt, an dem der Timer abläuft.

Mit Hilfe dieses Treibers soll nun ein Schrittmotor angesteuert werden. Der Treiber kann über den Pfad */dev/rtc* angesprochen werden. Wie bereits erwähnt, muss ein Signal auf den Parallelport gelegt werden. Durch den - bereits aus der Abbildung 10.4 bekannten - Treiber kann über die Funktion *write* ein Zeichen auf dem Parallelport ausgegeben werden. Dieser Treiber wird über den Pfad */dev/lpt* angesprochen.

**2. Aufgabe:** *Entwickeln Sie ein Programm, dass den oben beschriebenen Schrittmotor ansteuert. Die Anzahl der Schritte und die Geschwindigkeit sollen durch den Anwender bestimmt werden. Beachten Sie, dass die Geschwindigkeit nicht zwei Umdrehungen in der Sekunde übersteigen darf. Außerdem soll das Programm eine Option bieten, mit der bestimmt wird, dass die Anzahl der Schritte unendlich groß ist.*

Der Treiber aus Abbildung 11.4 hat den Nachteil, dass nur ein Prozess diesen Treiber verwenden darf.

**3. Aufgabe:** *Erweitern sie den Treiber aus Abbildung 11.4 so, dass mindestens zehn Prozesse diesen Treiber gleichzeitig verwenden dürfen. Um die Interrupt Response Time klein zu halten, sollte der Interrupt-Handler äußerst kurz sein. Erzeugen Sie einen Kernel-Thread, der die eigentliche Aufgabe des Interrupt-Handlers übernimmt. Dieser Thread wird vom Interrupt-Handler geweckt und überprüft, ob ein Timer abgelaufen ist und legt sich anschließend wieder schlafen. Die Priorität des Threads soll, wie im Abschnitt 5.3 beschrieben, durch die Priorität des Prozesses bestimmt werden, der die höchste Priorität aller Prozesse besitzt, die den Treiber geöffnet haben. Durch diese Vorgehensweise kann ein Prozess, der den entwickelten Treiber nicht verwendet, den Thread verdrängen, falls er eine höhere Priorität besitzt, als die Prozesse, die den Timer verwenden.*

## 11.4 4. Versuch: Regelung mit RTLinux

Ein Modell eines Regelkreises aus Fischertechnik soll nun mit Hilfe eines Reglers unter RTLinux angesteuert werden. Das Modell stellt einen Raum dar, der bei wechselnden Randbedingungen auf konstanter Temperatur gehalten werden soll. Geheizt wird mittels zweier Halogenlampen, beeinflusst wird die Temperatur durch einen Ventilator, eine verschiebbare Wand und eine Dachöffnung, sowie die natürlich Abstrahlung und Knovektion. Die Temperatur wird mit einem Sensor gemessen. Der steuernde Rechner ist mittels einer DA/AD Wandlerkarte mit dem Modell verbunden und kann somit Stellwerte auslesen und setzen. Es soll nun die Leistung der Heizung geregelt werden, so dass die Temperatur konstant bei 42 Grad Celsius bleibt. Ein externes Programm stört dabei den Temperaturverlauf durch Veränderung der anderen Stellgrössen. Der Regelalgorithmus kann frei gewählt werden. Der Fehler des Reglers, das heisst, die Abweichung von der Sollgrösse, soll über die Zeit aufgezeichnet werden.
Zu diesem Versuch werden weiter Informationen zu Beginn des entsprechenden Versuchsblocks ausgeteilt.

# Anhang A:Nützliche Funktionen für die Treiberprogrammierung

Funktionen aus der C-Library (libc, glibc) dürfen nicht im Kern bzw. in Treibern verwendet werden. Aus diesem Grund existieren im Kern einige Befehle, die die Befehle aus der C-Library ersetzen sollen. Allerdings sind diese Befehle nicht so komfortabel wie die aus der C-Library. Außerdem existieren nicht für alle Befehle der C-Library äquivalente Befehle im Kern. Es folgt ein Auszug der wichtigsten Befehle. Soweit nicht gesondert angegeben, gelten für RTLinux die gleichen Befehle wie bei Linux.

## 1.1 Ausgabe

**Linux:**
```
#include <linux/kernel.h>
void printk(char *format, ...);
```

**LynxOS:**
```
void cprintf(char *format, ...);
keine floats!
```

**QNX:** (kein Kernel!)
```
#include <stdio.h>
int printf(const char* format, ... );
```

Um die Möglichkeiten der bekannten *printf*-Funktion (z.B. für Kontroll- und Debug-Informationen) zu bieten, stellt der Kern diese Funktionen zur Verfügung. Durch den *syslog*-Daemon werden diese Ausgaben in eine Datei umgeleitet.

## 1.2 Speicher anfordern für Treibermodul

**Linux:**
```
#include <linux/malloc.h>
void* kmalloc(size_t size, int prio);
```

**LynxOS:**
```
char* sysbrk(long size);
```

**QNX:** (kein Kernel!)
```
#include <malloc.h>
void* malloc( size_t size );
```

Durch diese Funktion kann dynamischer Speicher (der Grösse *size* Bytes) angefordert werden. Bei Linux sollte man für *prio* das Makro GFP_KERNEL einsetzen. Für Echtzeitbedingungen eignet sich GFP_ATOMIC, was sparsam eingesetzt werden sollte.

Die Freigabe des angeforderten Speichers erfolgt durch diese Funktion:

**Linux:**
```
void kfree(void* addr);
```

**LynxOS:**
```
void sysfree(char* addr, long size);
```

**QNX:** (kein Kernel!)
```
void free( void* ptr );
```

## 1.3 Portoperationen

**Linux:**
```
#include <sys/io.h>
void outb(unsigned char data, int port);
unsigned char data=inb(int port);
```

**LynxOS:**
```
#include <port_ops_x86.h>
void __outb(int port, unsigned char data);
unsigned char data=__inb(int port);
```

**QNX:**
```
#include <hw/inout.h>
#include <sys/mman.h>
ThreadCtl( _NTO_TCTL_IO, 0 ); /* Rechte holen */
uintptr_t myport=mmap_device_io( size_t len, uint64_t port);
void out8(myport, uint8_t data );
uint8_t data=in8(uintptr_t myport);
```

Hiermit gibt man den Wert *data* auf dem IO-Port *port* aus, bzw liest vom Port in die Variable *data* ein. Bei QNX muss mit **ThreadCtl** dem Thread einmal das Recht erteilt werden, auf Hardware zuzugreifen. Dann muss der Port noch mit **mmap_device_io** eingemappt werden. Für diese beiden Aktionen muss der Prozess *root* Rechte besitzen.

## 1.4 Interrupts

**Linux:**
```
#include <asm/irq.h>
#include <linux/signal.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
int request_irq(int irq, void(*func)(int irq, void* dev_id, struct
pt_regs* regs), unsigned long irqflags, const char *devname, void
*dev_id);
...
int free_irq(int irq, void *dev_id);
```

**RTLinux:**
```
#include <rtl_core.h>
int rtl_request_irq (unsigned int irq,
unsigned int (*func)(unsigned int irq, struct pt_regs *regs));
...
```

```
int rtl_free_irq(unsigned int irq);
```

**LynxOS:**

```
int iointset(int irq + 32, int(*func)(), char *arg);
...
int iointclr(int irq + 32);
```

**QNX:**

```
#include <sys/neutrino.h>
ThreadCtl( _NTO_TCTL_IO, 0 ); /* Rechte fuer Hardwarezugriff */
int id=InterruptAttach( int irq, const struct sigevent * (*
func)(void *, int),const void * area, int size, unsigned flags );
...
int InterruptDetach(int id);
```

Diese Funktionen weisen den Interrupt *irq* dem Interrupt-Handler *func* zu, bzw. heben die Zuordnung auf. Tritt der Interrupt *irq* ein, bewirkt dieser die Ausführung der Funktion *func.* Linux: *irqflags* setzt man für schnelle ISR auf SA_INTERRUPT, *devname* taucht in /proc/interrupts als Name auf, *dev_id* unterscheidet Geräte bei Interrupt-Sharing, sonst (normalerweise) ist es NULL. Bei LynxOS muss zu der Interrupt-Nummer noch 32 addiert werden, die zusätzlichen Parameter *area*, *size*, *flags* bei QNX können fürs erste mit NULL, 0, 0 belegt werden. Falls der Thread noch keine Hardware-Privilegien hat, muss er sie mit **ThreadCtl** holen.

## 1.5 Ausschalten von Kontextwechseln und Interrupts.

**LynxOS:**

```
void sdisable(int ps);
/* kritische Region */

void srestore(int ps);
```

Nachdem die Funktion **sdisable** ausgeführt wurde, wird kein Kontext-Wechsel mehr durchgeführt. Der aktuelle Prozess kann nur durch einen Interrupt unterbrochen werden. Durch die Funktion **srestore** wird wieder ein Kontext-Wechsel ermöglicht. Zwischen diesen beiden Funktionen wird so eine kritische Region erzeugt. Die Dauer dieser kritischen Region sollte sehr kurz sein, da sonst die Antwortzeit der einzelnen Prozesse vergrößert wird.

**Linux:**

```
#include <asm/system.h>
unsigned long flags;
void save_flags(flags);
void cli(); /* Alle Interrupts gesperrt*/
/* kritische Region */

void sti();
void restore_flags(flags);
```

**RT-Linux:**

```
#include <rtl_sync.h>
void rtl_no_interrupts(rtl_irqstate_t state); /* Alle Interrupts */
/* kritische Region */                        /* gesperrrt */

void rtl_restore_interrupts(rtl_irqstate_t state);

#include <rtl_core.h>
int rtl_hard_disable_irq(unsigned int irq); /* Nur irq gesperrt*/
/* kritische Region */
int rtl_hard_enable_irq(unsigned int irq);
```

**LynxOS:**

```
void disable(int ps); /* Alle Interrupts gesperrt */
/* kritische Region */
void restore(int ps);
```

**QNX**:

```
#include <sys/neutrino.h>
ThreadCtl( _NTO_TCTL_IO, 0 );
memset( spinlock, 0, sizeof( *spinlock ) );
void InterruptLock( intrspin_t* spinlock ); /* Alle Int. gesperrt */
/* kritische Region */
void InterruptUnlock( intrspin_t* spinlock );
int InterruptMask( int irq, int id ); /* Interrupt irq gesperrt */
/* kritische Region */
int InterruptUnmask( int intr, int id );
```

Zwischen den beiden Funktionen werden die ankommenden Interrupts ignoriert. Werden alle Interrupts gespert, kann auch **kein Timer-Interrupt** bearbeitet werden. **ACHTUNG!** Da der Scheduler durch einen Timer-Interrupt aktiviert wird, kann dieser auch keine Rechenzeit bekommen und einem anderen Prozess eine neue Zeitscheibe zuteilen. Aus diesem Grund verhindern solche Funktionen wie **sdisable** und **srestore** einen Kontext-Wechsel. Zusätzlich werden aber auch noch Interrupts blockiert. Nach **restore** bzw. **sti** werden die in der Zwischenzeit angekommenen Interrupts behandelt.

## 1.6 Signale und Prioritäten für Nicht-Realzeit-Prozesse

**LynxOS**, **Linux**, **QNX**:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

Diese Funktion ermittelt die Id des aufrufenden Prozesses.

**LynxOS, Linux, QNX:**

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

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

Mit diesem Befehl wird das Signal *sig* zu dem Prozess mit der Id *pid* gesendet.

**LynxOS, Linux, QNX:**

```
#include <sys/resource.h>
int getpriority(PRIO_PROCESS, id_t pid);
int setpriority(PRIO_PROCESS, id_t pid, int value);
```

Diese Funktion ermittelt/setzt die Priorität auf der Ebene des Anwenders *who* des aktuellen Prozesses. Dies wirkt sich aber **nicht** bei POSIX-Scheduling nach SCHED_FIFO oder SCHED_RR aus! Zu den Scheduling-Eigenschaften nach POSIX siehe Kapitel 9.4.

Für Standard-Signale sieht die Zuweisung eines Handlers wie folgt aus. Hier wird dem Signal *signal* der Handler *handler* zugewiesen.

**LynxOS, Linux, QNX:**

```
#include <signal.h>
struct sigaction action, old_action;
void handler (int my_signal);
action.sa_handler = &handler;
action.sa_flags = 0;
sigaction(int signal, &action, &old_action);
```

## 1.7 Anmelden von Geräten und Treibern

Unter Linux gibt es Registrierfunktionen, die einen Gerätetreiber im Kern anmelden:

```
int register_chrdev(unsigned int major, const char* name,
struct file_operations *fops);
int register_blkdev(unsigned int major, const char* name,
struct file_operations *fops);
```

Diese Funktion registriert einen Treiber für zeichen- bzw. blockorientierte Geräte. Hierbei ist *major* die gewünschte Major-Nummer des Gerätes und *fops* die Adresse der file_operations-Struktur. Mit *name* wird der symbolische Name (z.B. „tty“, „lp“) angegeben. Die Abmeldung einer Treiber-Gerätezuordnung erfolgt durch:

```
int unregister_chrdev(unsigned int major, const char* name);
int unregister_blkdev(unsigned int major, const char* name);
```

Installiert und deinstalliert wird der Treiber durch die Befehle **insmod** *objname* bzw. **rmmod** *objname*. Über /dev einträge, welche mit **mknod** *name [c|b] major minor* erzeugt werden, kann man mit dem Gerät kommunizieren.

**LynxOS**:

Hier müssen sich Treiber nicht anmelden, dies ist bei der Installation von Hand zu erledigen.

Mit **drinstall** *-[c|b] objname* wird der Treiber *objname* für char/block Geräte installiert und eine Treibernummer zurückgegeben. Der Befehl **drivers** zeigt eine Liste der installierten

Treiber. Mit **devinstall** *-[c|b] -d Treibernummer Geräteinfo* wird der Treiber dem Gerät mit *Geräteinfo* zugeordnet und eine major-Nummer zurückgegeben. Der Befehl **devices** zeigt alle aktive zuordnungen. Dann ist mit **mknod** wie oben ein /dev Eintrag zu erstellen. Zum deinstallieren des Treibers ist erst die Zuordnung zu löschen: **devinstall** *-u -[b|c] major-Nummer*. Danach wird der Treiber mit **drinstall** *-u -[b|c] Treibernummer* entfernt. Bei Neuinstallation auf passende major-Nummer achten!

**QNX**

QNX ist wieder etwas anders:

```
#include <sys/dispatch.h>
int resmgr_attach
       ( dispatch_t * dpp,
         resmgr_attr_t * attr,
         const char * path,
         enum _file_type file_type,
         unsigned flags,
         const resmgr_connect_funcs_t * connect_funcs,
         const resmgr_io_funcs_t * io_funcs,
         RESMGR_HANDLE_T * handle );
```

Statt Treibern gibt es hier Resource-Manager. Die Resource-Manager sind User-Space Programme, sie werden aufgerufen wie gewöhnliche Programme. Mit obigem Befehl wird ein Name *path* angemeldet und Funktionen *connect_funcs*, bzw. *io_funcs* für Zugriffe auf das Gerät definiert. Wegen des Umfang des Themas wird auf die ausführliche QNX-Dokumentation verwiesen.

# Anhang B:Eine kurze Beschreibung von Gnuplot

In diesem Skript wird unter anderem erläutert, wie Latenzzeiten und Bandbreiten gemessen werden. Um die Ergebnisse besser deuten zu können, werden sie meistens graphisch dargestellt. Es existieren eine Reihe von Programmen, mit dessen Hilfe Messergebnisse graphisch dargestellt werden können. Einen Klassiker unter diesen Programmen stellt *Gnuplot* dar, das sehr klein und einfach zu bedienen ist. Mit Hilfe von Gnuplot können 2- und 3-dimensionale Grafiken erstellt werden. In dieser kurzen Beschreibung von Gnuplot wird erläutert, wie aus Messergebnissen 2-dimensionale Bilder erzeugt werden. Eine ausführlichere Dokumentation des Programmes findet man im Internet unter *http://www.cs.dartmouth.edu/gnuplot_info.html*.

Gestartet wird Gnuplot mit dem Befehl:

**gnuplot**

Optional kann beim Aufrufen auch eine Datei als Parameter übergeben werden. Gnuplot führt dann die darin enthaltenen Kommandos aus. Eine solche Datei könnte z.B. folgendermaßen aussehen:

```
set terminal postscript eps color 17
set output "GnuplotExample.eps"

set xlabel "x-Achse"
set ylabel "y-Achse"

plot "GnuplotData1" title 'f(x)=log(x)' with linespoints,
'GnuplotData2' title 'g(x)=x' with linespoints
```

**Abbildung 2.1:** Eine Beispieldatei mit Gnuplot-Befehlen

Der Aufruf „gnuplot example.gplot“ erzeugt dann die in der Abbildung 2.2 dargestellte Postscript-Datei „GnuplotExample.eps“. Bei dieser Datei handelt es sich um eine spezielle Postscript-Form (encapsulated postscript = eps), die sich besonders gut in Text einbinden lässt.

Die hierbei benutzten Werte sind in den Abbildungen 2.3 und 2.4 dargestellt. Diese Werte müssen unter den Namen „GnuplotData1“ bzw. „GnuplotData2“ im gleichen Verzeichnis abgespeichert werden, in dem sich auch die Datei aus Abbildung 2.1 befindet. Wurde Gnuplot ohne Parameter aufgerufen, so befindet man sich nun im Kommandozeileneditor und muss die Befehle interaktiv eingeben. Im Folgenden werden die wichtigsten Kommandos kurz erläutert.

## 2.1 Das plot-Kommando

```
plot {ranges} {<function> | {"<datafile>" {using ...}}}
{title} {style} {, <function> {title} {style}...}
```

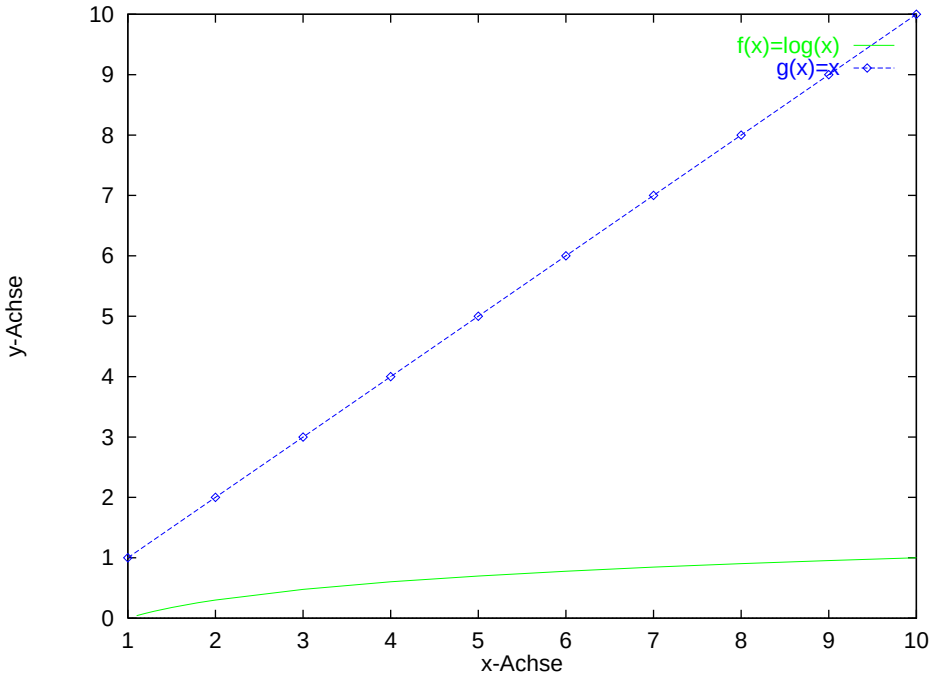



**Abbildung 2.2:** Das von Gnuplot erzeugte Bild

```
1.1 0.041392685
1.2 0.079181246
1.3 0.113943352
1.4 0.146128035
1.5 0.176091259
1.6 0.204119982
1.7 0.230448921
1.8 0.255272505
1.9 0.278753601
2.0 0.301029995
3.0 0.477121254
4.0 0.602059991
5.0 0.698970004
6.0 0.77815125
7.0 0.84509804
8.0 0.903089987
9.0 0.954242509
10.0 1
```

**Abbildung 2.3:** Beispielwerte aus der Datei „GnuplotData1“

Die Bereichsangabe (range) kann benutzt werden, wenn von einer Funktion nur ein bestimmter x-Achsen-Abschnitt berechnet werden soll. Beispielsweise erzeugt das Kommando ‘plot [x=0:2*pi] [-1:1] sin(x)’ die Sinus-Kurve im Bereich [0:2pi].

```
1.0 1.0
2.0 2.0
3.0 3.0
4.0 4.0
5.0 5.0
6.0 6.0
7.0 7.0
8.0 8.0
9.0 9.0
10.0 10.0
```

**Abbildung 2.4:** Beispielwerte aus der Datei „GnuplotData2“

Beim plot-Kommando ist die Angabe einer Funktion (bereits vordefinierte oder selbstdefinierte Funktionen) genauso möglich, wie die Angabe einer Datei, aus der die Werte ausgelesen werden. Mit einer Formatangabe nach dem Schlüsselwort ‘using’ können auch ganz bestimmte Daten aus der Eingabedatei herausgefiltert werden.

Sollen mehrere Kurven in demselben Diagramm gezeichnet werden, so sind die entsprechenden Kommandos durch Komma voneinander zu trennen. Über das Schlüsselwort ‘title’ kann für jede Kurve ein Bezeichner benannt werden, der in der Legende des erzeugten Bildes verwendet werden soll.

Das Aussehen der Kurven (style) kann durch eine der folgenden Möglichkeiten gewählt werden:

```
with lines|points|linespoints|impulses|dots|steps|errorbars
```

Wird zum Beispiel *linespoint* ausgewählt, so werden die einzelnen Werte durch Punkte markiert, und diese Punkte werden noch zusätzlich durch eine Linie miteinander verbunden. In Abbildung 2.2 wurde beispielsweise diese Kurvenart ausgewählt.

## 2.2 Das set-Kommando

Die Ausgabeform der Zeichnung kann in vielerlei Hinsicht den eigenen Wünschen angepasst werden. Dafür gibt es den set-Befehl, der die Werte für die aktuelle Zeichnung festlegt.

Hier einige Beispiele:

- **terminal** Hiermit kann der Typ des Ausgabegeräts festgelegt werden. Beispielsweise führt ‘set terminal X11’ zur Ausgabe der Grafik auf den Bildschirm und ‘set terminal postscript eps color 17’ in Verbindung mit ‘set output „outputfile.eps“‘ zur Erzeugung einer eps-Datei. Durch *color* und *17* wird angegeben, dass das zu erzeugende Bild farbig sein soll und die Schriftgröße 17 Punkte beträgt.
- **title** Die Überschrift über dem Bild kann hiermit gesetzt werden. (Nicht zu verwechseln mit dem title vom plot-Befehl.)
- **label** An jeder beliebigen Stelle in der Grafik kann ein Text eingefügt werden.

Zusammen mit **arrow** können so Erklärungen in der Grafik erscheinen.
- **grid** Die Kurve oder das Messblatt können mit einem Gitter unterlegt werden.

## 2.3 Das load-Kommando

Sind mehrere Gnuplot-Kommandos in eine Datei geschrieben worden (siehe Abbildung 2.1), kann diese mit dem load-Kommando eingelesen werden. Die in der Datei enthaltenen Kommandos werden anschließend nacheinander ausgeführt.

## 2.4 Das help-Kommando

Neben der oben angegebenen Internet-Seite kann auch das Gnuplot-Hilfe-System benutzt werden, um genauere Informationen zu erhalten. Z.B. erhält man mit „help expression functions real“ eine Liste der eingebauten Funktionen.

# Anhang C:Der GNU C Compiler

Der GNU C Compiler kann C und C++ Programme übersetzen. Der Sourcecode muss dazu die Endung .c (C Programme) bzw. .C|.cc|.cpp (C++ Programme) haben.

Der Befehl zum Übersetzen eines C bzw. C++ Programmes lautet:

**gcc [option | filename]...**

**g++ [option | filename]...**

Wird der Compiler ohne eine Option nur mit dem Programmnamen aufgerufen, so wird das Programm übersetzt, mit den entsprechenden Bibliotheken gelinkt und die ausführbare Programmdatei „a.out“ angelegt.

Das Compilieren ist ein komplexer Vorgang, der mehrere Zwischenschritte umfasst und bei dem einige temporäre Dateien als „Zwischenprodukte“ erzeugt werden. Durch geeignete Optionen kann der GCC dazu gebracht werden, die erforderlichen Schritte einzeln durchzuführen.

Hier nun eine kurze Zusammenfassung der wichtigsten Optionen. Mehr Informationen hierzu findet man in den Manpages des GCC.

| Option | Wirkung |
|---|---|
| -c | Das angegebene Programm wird nur übersetzt und die Objektdatei (Endung .o) erzeugt. |
| -o Name | Die ausführbare Datei erhält den angegebenen Namen und nicht wie üblich den Namen ‘a.out’. |
| -g | Es wird zusätzliche, zum symbolischen Debuggen notwendige Information in die Objektdatei hineingeschrieben. |
| -pg | Es wird beim Compilieren zusätzliche Information hinzugefügt, die der Profiler *gprof* benötigt, um das Programm zu analysieren. |
| -Wall | Es ist die höchste Warnstufe eingestellt und es wird genauestens auf die Einhaltung der Syntax geachtet. |
| -Ox | x ist hierbei eine der Zahlen 1, 2 oder 3. Es wird die entsprechende Optimierungsstufe eingestellt. |
| -M | Der Compiler erzeugt Informationen, um die Dependencies jeder Objektdatei erzeugen zu können. |
| -I Pfad | Das angegebene Verzeichnis wird in den Include-Path aufgenommen. Hier wird dann auch nach Header-Dateien gesucht. |
| -L Pfad | Hier kann ein zusätzlicher Library-Path gesetzt werden. |
| -llibname | Mit dieser Option können verwendete Libraries angegeben werden. |

Wurde mit der Option -c eine Objektdatei erzeugt, so wird das lauffähige Programm einfach mit „gcc objektdatei.o -o programmname“ erzeugt.

# Anhang D:Der GNU-Profiler *gprof*

Mit dem GNU-Profiler kann man sich die von Funktionen verbrauchte CPU-Zeit anzeigen lassen. Hieraus erkennt der Entwickler, wo das Programm wieviel Zeit benötigt und welche Funktionen welche anderen Funktionen aufrufen.

Dazu muss das Programm mit der Option -pg übersetzt werden. Zur Laufzeit wird dann in die Datei gmon.out geschrieben, wieviel Zeit wo verbraucht wurde. Der Profiler liest diese Datei ein und stellt die Information aufbereitet dar. Das kann helfen, fehlerhafte oder langsame Stellen im Programm zu finden und zu beseitigen.

Das Resultat von GPROF ist ein Textfile mit zwei Tabellen: das ‘Flat Profile’ und der ‘Call Graph’. Das ‘Flat Profile’ zeigt an, wieviel Zeit die jeweiligen Funktionen benötigten und wie oft sie aufgerufen wurden. Der ‘Call Graph’ zeigt für jede Funktion, welche Funktion sie aufrief, welche anderen Funktionen sie aufruft und wie oft dies passiert.

Im Folgenden nun eine kurze Übersicht der wichtigsten Schritte und Kommandos:

Um die für GPROF nötigen Informationen zu erzeugen, muss das Programm mit der **Option -pg** compiliert werden. Im Anschluss daran muss das Programm wie gewöhnlich gestartet werden, damit die Profiler-Informationen erzeugt werden. Diese werden in das File ‘gmon.out’ geschrieben. Allerdings ist es nur korrekt und brauchbar, wenn der Programmlauf korrekt mit ‘return’ oder ‘exit’ beendet wurde.

Der Aufruf des Profilers kann nun folgendermaßen erfolgen:

**gprof options [executable-file [profile-data-files...]] [> outfile]**

Wird hierbei kein ‘executable-file’ angegeben, so versucht GPROF, das File ‘a.out’ zu benutzen. Ebenso wird anstelle der ‘profile-data-files’ das Standard-File ‘gmon.out’ benutzt. Mit den verschiedenen Options hat man u.a. die Möglichkeit, das Resultat und die Ausgabe von GPROF zu beeinflussen. Z.B. kann die Ausgabe bestimmter Funktionen unterdrückt werden. Näheres hierzu ist den Manpages zu entnehmen.

Zum besseren Verständnis des Resultats hier nun ein Beispiel mit den entsprechenden Erläuterungen:

Das ‘Flat Profile’ zeigt die Gesamtsumme, wieviel Zeit jede Funktion benötigte. Hier ein Ausschnitt des ‘Flat Profiles’ eines kurzen Programmes:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
```

```
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

Die erste Spalte ‘% time’ gibt Auskunft darüber, wieviel Prozent der Gesamtlaufzeit die jeweilige Funktion benötigte. ‘cummulative seconds’ gibt an, wieviel Zeit der Rechner insgesamt brauchte, um die jeweiligen Funktionen auszuführen (die Werte werden alle aufaddiert). ‘self seconds’ ist die Zeit, die jede einzelne Funktion benötigte. Die Anzahl, wie häufig eine Funktion aufgerufen wurde, steht in der Spalte ‘calls’. Die nächsten beiden Spalten geben Auskunft darüber, wieviele Millisekunden die Funktionen im Schnitt pro Aufruf benötigten. Am Ende ist der Name der jeweiligen Funktion angegeben.

Der ‘Call Graph’ gibt an, wieviel Zeit jede Funktion und die dazugehörigen Unterfunktionen benötigten. Hierbei sind die einzelnen Funktionen und ihre Funktionsaufrufe durch die Linien sichtbar voneinander getrennt.

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 sec-
onds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
               0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
```

```
-----------------------------------------------
[4]     59.8    0.01       0.02       8+472  <cycle 2 as a whole>   [4]
                0.01       0.02     244+260      offtime <cycle 2> [7]
                0.00       0.00     236+1         tzset <cycle 2> [26]
```

Die Zeile, in der am Anfang ein Index (für jede Funktion ein Index) steht, gibt an, welche Funktion in diesem Abschnitt genauer betrachtet wird. Alle anderen Zeilen enthalten die Funktionen, die von dieser Funktion aufgerufen werden. Die Spalte ‘% time’ gibt an, wieviel Prozent der Gesamtlaufzeit diese Funktion (inkl. der aufgerufenen Funktionen) benötigt hat. Die nächste Spalte beinhaltet die Gesamtzeit, die diese Funktion benötigte und daran anschließend steht die Gesamtzeit der aufgerufenen Funktionen. Die Spalte ‘called’ gibt an, wie oft diese Funktion aufgerufen wurde. Tritt hier eine Rekursion auf, so folgt ein ‘+’ gefolgt von der Anzahl der rekursiven Aufrufe.

Wurde diese im Abschnitt erläuterte Funktion von einer anderen Funktion aufgerufen, so steht die rufende Funktion in der Zeile darüber. Alle aufgerufenen Funktionen folgen anschließend. Die Spalten haben hier eine andere Bedeutung als in der gerade erläuterten Zeile. Genaue Erläuterungen der jeweiligen Spalte erzeugt GPROF im Anschluss an den ‘Call Graph’. Gleiches gilt auch für das ‘Flat Profile’.

# Anhang E:Verwendung von GNU Make

Das *Makefile* unterstützt den Programmierer bei der Erstellung größerer Programmpakete. Bei einem komplexen Programm ist im Allgemeinen der Sourcecode auf mehrere Dateien verteilt und muss aber bei der Compilierung mit verschiedenen Libraries verknüpft werden. Wie dies geschehen soll und in welchen Verzeichnissen sich die einzelnen Elemente befinden, wird im Makefile angegeben. Dabei stellt das Makefile eine Beschreibung dar, welches den Compilierungsablauf steuert.

Das Prinzip von GNU Make beruht darauf, dass in einer sogenannten *Beschreibungsdatei (Makefile)* schematisch beschrieben ist, was *(Ziel)* woraus *(Voraussetzungsdateien)* wie *(Rezept)* zu machen ist. Eine dementsprechende Zusammenstellung von Ziel, Voraussetzungsdateien und Rezept heißt *Regel*. Die schematische Form, wie eine derartige Regel im Makefile stehen muss, ist wie folgt:

```
ziel: voraussetzungsdateien
<tab> rezept
```

Hierbei ist `ziel` das zu erzeugende Objekt, `voraussetzungsdateien` eine durch Leerzeichen getrennte Liste von Dateien und `rezept` der UNIX-Befehl, mit dem aus den Voraussetzungsdateien das Ziel erzeugt werden kann. Sind zur Erzeugung des Zieles mehrere Befehle erforderlich, so muss jeder dieser Befehle jeweils in einer neuen, mit einem Tabulator beginnenden Zeile aufgeführt sein.

Ein Makefile kann mehrere derartige Regeln enthalten, die von GNU Make wie folgt behandelt werden:

1. Zunächst wird jede Voraussetzungsdatei auf Aktualität überprüft, d.h.
   - ist die Voraussetzungsdatei selbst das Ziel einer anderen Regel im Makefile, so wird zunächst diese Regel für die Voraussetzungsdatei behandelt.
   - ist für die Voraussetzungsdatei keine Regel vorhanden, so muss die Voraussetzungsdatei existieren. (Andernfalls wird das make-Kommando mit einer entsprechenden Fehlermeldung `don't know how to make ...` abgebrochen.)

2. Nun sind alle Voraussetzungsdateien vorhanden und aktuell. Jetzt wird überprüft, ob eine der Voraussetzungsdateien „neuer“ ist als das aktuelle Ziel.
   - Ist dies der Fall, so werden alle im Rezept angegebenen Befehle zur Ausführung gebracht.
   - Ist dies nicht der Fall, so ist das Ziel bereits aktuell und die Befehle brauchen nicht ausgeführt werden.

## 5.1 Statische Makros

Im Makefile können Makros für Zeichenketten (z.B. Pfade und Dateien) definiert und diese Makros im Makefile an unterschiedlichen Stellen verwendet werden. Die Definition und Verwendung von Makros ist ähnlich zu der von Shell-Variablen:

```
CC = gcc
```

Der Aufruf sieht dann z.B. folgendermaßen aus:

```
$(CC)
```

## 5.2 Dynamische Makros

Neben den gerade erwähnten statischen Makros gibt es folgende Laufzeitmakros:

| Makro | Bedeutung |
|---|---|
| $@ | aktuelles Ziel (kompletter Name) |
| $* | aktuelles Ziel (Name ohne Endung) |
| $< | aktuelle Voraussetzungsdatei; erste in der Liste, welche neuer als das aktuelle Ziel ist |
| $? | Liste aller aktuellen Voraussetzungsdateien, die neuer als das aktuelle Ziel sind |

## 5.3 Explizite Regeln

Unter einer expliziten Regel versteht man die zu Beginn erläuterte Form. Sie gibt an, was woraus wie zu machen ist.

## 5.4 Implizite Regeln

Bei der Erstellung eines Projektes ist es aber im Allgemeinen so, dass viele unterschiedliche, aber gleichartige Vorgänge durchzuführen sind. So muss etwa aus jedem Quelltext eine zugehörige Objektdatei erzeugt werden. Um nicht für jeden Quelltext eine eigene Regel schreiben zu müssen, kann man sich die dem make bekannten Standard-Dateiendungen in sogenannten impliziten Regeln wie folgt zunutze machen:

```
.cc.o:
<tab> $(CC) -c $<
```

Diese Regel beschreibt, wie generell aus einer Datei mit der Endung .cc eine Datei mit der Endung .o erzeugt werden kann. (Ziel ist eine beliebige Datei mit Endung .o, wobei die Voraussetzungsdatei den gleichen Namen, aber die Endung .cc hat.)

## 5.5 Beispiel

Zuerst werden einige Makros für die Pfade und Dateien definiert. Anschließend werden die Pfade für die Include-Files und Libraries angegeben. Weiterhin werden einige Optionen definiert. Die Option -g bedeutet z.B., dass der Compiler Informationen in den Programmcode einbindet, die für den Debugger wichtig sind. Die Option -02 bringt den Compiler dazu, einen geschwindigkeitsoptimierten Code zu erzeugen. Der anschließende Teil ist der wichtigste Teil des Makefiles: hier werden die Regeln und Abhängigkeiten definiert, die dem Compiler genau angeben, wie er mit den einzelnen Programmteilen umzugehen hat. Zusätzlich wird eine Funktionalität des GNU-Compilers benutzt, die dazu dient, von einer oder mehreren Quelldateien automatisch die Abhängigkeiten zu erkennen. Das Kommando `make depend` veranlasst ein Durchsuchen der Quelldateien und fügt für jede gefundene `#include ...` Direktive einen entsprechen-

den Eintag in einer gesonderten Datei (.depend) ein.

```
PRJ_DIR  = .
PRJ_INC  = -I.

TARGET   = $(PRJ_DIR)/Beispiel

C_FILES  =
CC_FILES = Klasse1.cpp \
           Klasse2.cpp \
           Klasse3.cpp \
           main.cpp

O_FILES = $(patsubst %.cc, %.o, $(filter %.cc, $(CC_FILES)))
O_FILES += $(patsubst %.cpp, %.o, $(filter %.cpp, $(CC_FILES)))
O_FILES += $(patsubst %.C, %.o, $(filter %.C, $(CC_FILES)))
O_FILES += $(patsubst %.c, %.o, $(filter %.c, $(C_FILES)))

CC  = gcc
CXX  = g++
RM        = rm -f
DEBUG     = -g
CFLAGS  = -Wall -O2
CPPFLAGS  = $(CFLAGS)
INCLUDE   = $(PRJ_INC)
LDLIBS    = -lstdc++ -lm

# implicit rules C++ files
%.o : %.cc
        $(CXX) -c $(CPPFLAGS) $(INCLUDE) -o $@ $<
%.o : %.cpp
        $(CXX) -c $(CPPFLAGS) $(INCLUDE) -o $@ $<
%.o : %.C
        $(CXX) -c $(CPPFLAGS) $(INCLUDE) -o $@ $<

# implicit rules for C files
%.o : %.c
        $(CC) -c $(CFLAGS) $(INCLUDE) -o $@ $<

default: $(TARGET)

all:     $(TARGET)

$(TARGET): $(O_FILES)
        $(CC) -o $@ $(O_FILES) $(LDLIBS)
```

```
depend:

        $(CC) -M $(C_FILES) $(CC_FILES) $(CPPFLAGS) \
             $(INCLUDE) > .depend

clean:
        $(RM) $(O_FILES) $(TARGET) .depend core *~

-include         .depend
```

**Abbildung 5.1:** Beispiel für ein Makefile

# Anhang F:Eine Kurzanleitung des GNU Debuggers

Der GNU Debugger (kurz: GDB) bietet, grob gesagt, vier Möglichkeiten, Programmfehler zu entdecken:

- Der GDB startet das Programm und spezifiziert alles, was den Programmablauf beeinflussen könnte.
- Stoppt den Programmablauf an festgelegten Stellen (sogenannte Breakpoints).
- Gibt Informationen über das gestoppte Programm.
- Änderungen können durchgeführt werden, um so den Fehler zu beheben. Anschließend kann mit dem nächsten Fehler fortgefahren werden.

Um in einem Programm einen Fehler mit Hilfe des GDB zu finden, ist folgender Aufruf nötig:

**gdb <Programmname>**

Weitere Parameter für den Aufruf, die jedoch selten erforderlich sind, erhält man mit:

**gdb -h**

Die während des Debuggens am häufigsten benötigten Befehle sind die folgenden:

**break [<Dateiname>:] <Funktionsname>**
Hiermit wird ein Breakpoint an den Anfang der entsprechenden Funktion gesetzt, d.h. das Programm wird nach dem Starten bis zum ersten Eintritt in die Funktion abgearbeitet und dann angehalten. Es können auch mehrere Breakpoints gesetzt werden; dann hält das Programm an dem zuerst erreichten Breakpoint an.

**break [<Dateiname>:] <Zeilennummer>**
Hiermit wird ein Breakpoint zu Beginn der entsprechenden Zeile gesetzt.

**clear <Funktionsname/Zeilennummer>**
Hiermit wird ein mit ‘break’ erstellter Breakpoint wieder entfernt.

**run [<Argumente>]**
Startet das Programm. Wird das Programm normalerweise mit Parametern aufgerufen, so sind diese hinter dem ‘run’ aufzuführen. Es wird dann erst bei Erreichen eines Breakpoints angehalten.

**cont**
Nachdem das Programm durch Erreichen eines Breakpoints angehalten wurde, kann es mittels ‘cont’ fortgesetzt werden, bis der nächste Breakpoint erreicht wird.

**step [<Schrittzahl>]**
(step into) Nachdem ein Breakpoint erreicht wurde, kann das Programm zeilenweise

abgearbeitet werden. Mit ‘step’ wird genau eine Zeile des Programms abgearbeitet. Wird eine Zahl als Parameter übergeben, so wird ‘step’ entsprechend oft ausgeführt. Sollte bei einem Schritt eine Unterroutine aufgerufen werden, so wird der Code der Unterroutine verfolgt.

**next [<Schrittzahl>]**

(step over) Genau wie bei ‘step’, nur dass im Falle eines Funktionsaufrufs der Code der Unterroutine nicht verfolgt wird, sondern im folgenden Schritt die Zeile nach dem Aufruf der Unterroutine ausgeführt wird.

**finish**

(step out) Wurde beispielsweise mit ‘step’ in eine Unterroutine gesprungen, so kann diese durch ‘finish’ komplett abgearbeitet werden, d.h. das Programm wird erst nach Verlassen dieser Unterroutine wieder angehalten.

**backtrace**

Gibt den aktuellen Calling-Stack aus.

**print <Ausdruck>**

Gibt den Wert des angegebenen Ausdrucks aus. Dabei angesprochene Variablen müssen entweder global deklariert sein oder sich in derselben „Stackebene“ befinden. D.h. aus einer Unterroutine kann z.B. nicht der Wert einer Variable ausgegeben werden, die in der Routine deklariert ist, die die Unterroutine aufgerufen hat.

**display <Ausdruck>**

Prinzipiell wie ‘print’, nur dass der Wert des Ausdrucks jedes Mal ausgegeben wird, wenn das Programm anhält, sei es durch Erreichen eines Breakpoints oder nach dem Ausführen von ‘step’, ‘next’ oder ‘finish’.

**undisplay [<Nummer eines Ausdrucks>]**

Ein Ausdruck wird hiernach nicht mehr jedesmal ausgegeben, wenn das Programm anhält. Bei der Ausgabe des Wertes eines Ausdrucks (durch ‘print’ oder ‘display’) wird immer die eindeutige Nummer des Ausdrucks mit ausgegeben. Dieser muss als Parameter an ‘undisplay’ übergeben werden. Wird kein Parameter angegeben, so werden alle Ausdrücke vom ‘display’ entbunden.

**list [<Zeilennummer>]**

Mit ‘list’ werden zehn Zeilen des Programms ausgegeben. Wird eine Zeilennummer als Parameter übergeben, so werden die zehn Zeilen ab eben dieser Stelle ausgegeben. Wird kein Parameter übergeben, so werden zehn Zeilen um die Stelle des Programms herum ausgegeben, an der das Programm gerade hält. Weitere Parameter sind mit ‘help list’ zu erhalten.

**info <Diverse>**

Mit ‘info’ kann alles ausgegeben werden, was wichtig ist. Mit ‘info break’ können alle Breakpoints aufgelistet werden, mit ‘info display’ können alle Ausdrücke ausgegeben werden, deren Werte bei jedem Halt ausgegeben werden, etc. Eine Liste aller Möglichkeiten wird mit ‘help info’ gegeben.

**help [Befehlsklasse/Befehl]**

Ausführliche Hilfe. Wird an ‘help’ kein Parameter übergeben, so werden alle Befehlsklassen ausgegeben. Wird eine Befehlsklasse übergeben, so werden alle zur Klasse gehörenden Befehle ausgegeben. Wird ein Befehl übergeben, so erfolgt eine Beschreibung dieses Befehls.

**quit**

Beendet den GDB.

# Anhang G:Der Data Display Debugger

Der DDD ist eine graphische Bedienoberfläche für den rein zeilenorientierten GDB. Neben den gewöhnlichen Front-End Features hat der DDD ein graphisches Datendisplay (siehe Abbildung 7.1), in dem Datenstrukturen als Graphen dargestellt werden.

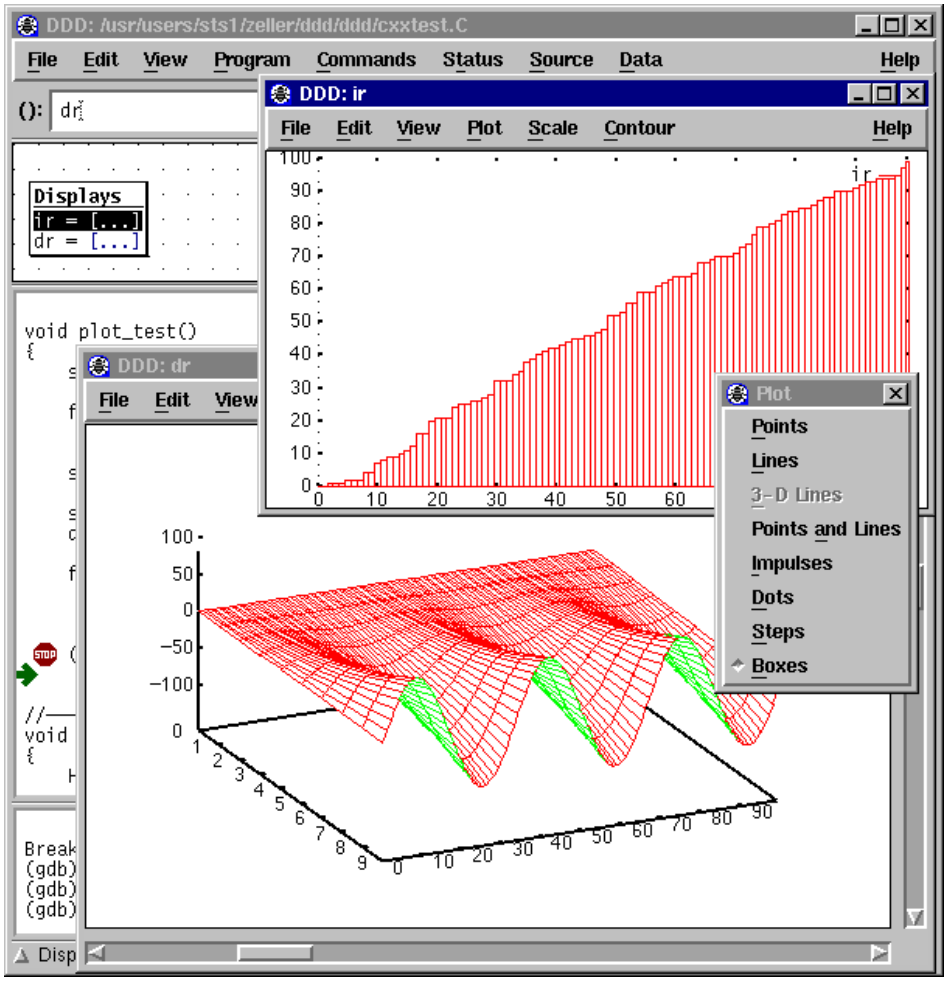



**Abbildung 7.1:** Graphische Darstellung eines Feldes

Per Mausklick können die aktuellen Datenstrukturen in einem graphischen Datendisplay angezeigt werden. Es werden Pointer dereferenziert, der Inhalt von Strukturen übersichtlich dargestellt (z.B. Bäume) und vieles mehr. Dabei werden die Werte der angezeigten Variablen bei jedem Programmstopp auf den aktuellen Stand gebracht, so dass Veränderungen der Werte leicht erkennbar sind.

Gestartet wird der DDD durch folgenden Aufruf:

**ddd**

Anschließend erscheint das Fenster aus der Abbildung 7.2. Unter dem Menü „File“ kann nun das zu untersuchende Programm geöffnet werden. Zusätzlich zu der Menüleiste und den Buttons (hier befindet sich z.B. der Button zum Setzen oder Löschen eines Breakpoints) erscheint das DDD Command Tool mit allen Debugging-Funktionen. Nach dem

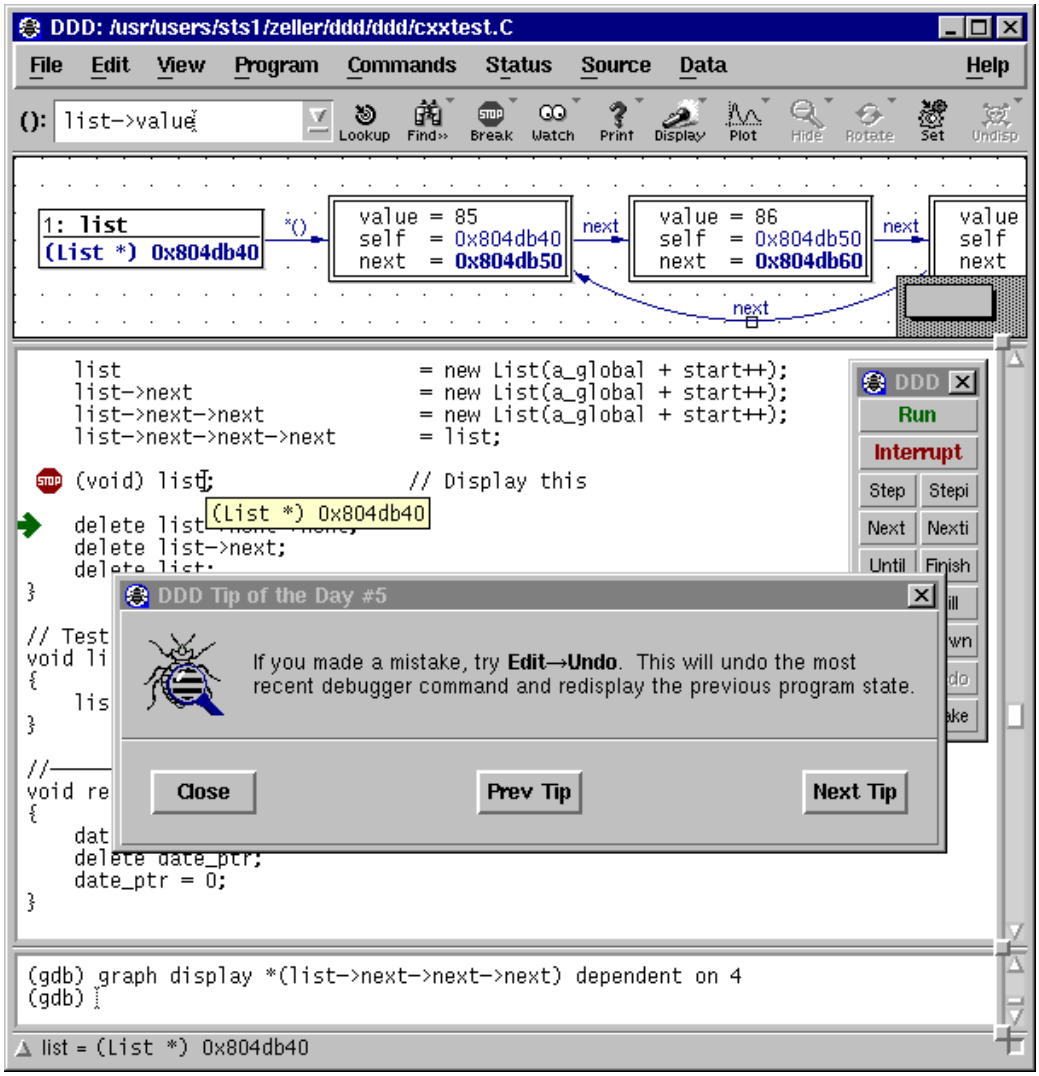



**Abbildung 7.2:** Eröffnungsfenster des DDD

Öffnen des Programms muss als nächstes festgelegt werden, an welchen Stellen das Programm angehalten werden soll. Dies geschieht durch Setzen eines Breakpoints. Anschließend kann der Start des Programms bzw. des Debuggerlaufs durch ‘Run’ erfolgen.

Im unteren Fenster (dies ist das GDB Textfenster) wird angezeigt, welche Befehle ausgeführt werden, und es gibt z.B. Auskunft über die mit ‘Print’ ausgewählten Variablen. Hier ist es auch möglich, die GDB-Befehle (siehe vorheriges Kapitel) direkt einzugeben.

Das Programm wird bis zu der Stelle ausgeführt, wo der Breakpoint gesetzt wurde. Von hier kann nun mit „Next“ die jeweils nächste Programmzeile ausgeführt werden. Werden im Programm Ein- oder Ausgabeoperationen ausgeführt, so werden diese über das GDB-Fenster realisiert.

Im Folgenden nun eine kurze Erläuterung einiger wichtiger Debugging Funktionen:

**Run**

Start des Programms

**Interrupt**
Unterbrechung des Programms

**Step**
Ausführung der nächsten Programmzeile. Ist es ein Funktionsaufruf, so wird in die Funktion hineingesprungen.

**Next**
Ausführung der nächsten Programmzeile. Funktionsaufrufe werden in einem ausgeführt.

**Finish**
Programmausführung wird fortgesetzt, bis die aktuelle Funktion beendet ist.

**Cont**
Programmausführung wird fortgesetzt. Das Programm stoppt erst am Programmende oder beim Erreichen des nächsten Breakpoints.

**Display**
Erzeugung eines Kästchens im oberen Teilfenster (graphisches Daten-Display). Inhalt ist der Name der ausgewählten Variable mit entsprechendem Wert. Bei jeder Änderung des Wertes wird das Kästchen aktualisiert. (Die Kästchen können beliebig verschoben werden)

**Print**
Ausgabe der ausgewählten Variable im GDB Textfenster

# Literaturverzeichnis


[1] T. Bemmerl*: Betriebssysteme I, Vorlesungsskript,* Lehrstuhl für Betriebssysteme, RWTH Aachen, 1997

[2] T. Bemmerl: *Betriebssysteme II, Vorlesungsskript,* Lehrstuhl für Betriebssysteme, RWTH Aachen, 1998

[3] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner: *Linux-Kernel-Programmierung - Algorithm und Strukturen der Version 2.0*, 4. Auflage, Addison-Wesley, 1997

[4] A. Cockcroft, R. Pettit: *Sun Performance and Tuning - Java and the Internet*, Second Edition, Sun Microsystems Press, 1998

[5] D. A. Curry: *UNIX Systems Programming for SVR4*, O'Reilly & Associates, Inc., 1996

[6] B. O. Gallmeister: *POSIX.4 - Programming for the Real World*, O‘Reilly & Associates, Inc., 1995

[7] S. Kleiman, D. Shah, B. Smaalders: *Programming With Threads*, Prentice-Hall International, Inc., 1995

[8] B. Kuhn: *Zeitgenau*, Artikel in der Zeitung „Linux-Magazin“, Ausgabe 11/98, Seite 34-42, Linux-Magazin Verlag, 1998

[9] D. Lewine: POSIX Programmer’s Guide, O’Reilly & Associates, Inc., 1991

[10] B. Lewis, D. J. Berg: *Threads Primer - A Guide to Solaris Multithreaded Programming*, Prentice-Hall International, Inc., 1995

[11] M. Loukides, A. Oram: Programmieren mit GNU Software, O’Reilly & Associates, Inc., 1997

[12] Lynx Real-Time Systems: *Writing Device Drivers for LynxOS*, 1998

[13] Lynx Real-Time System: *LynxOS User’s Guide*, 1998

[14] B. Nichols, D. Buttlar, J. P. Farrell: *Pthreads Programming*, O‘Reilly & Associates, Inc., 1996

[15] QNX Software Systems Ltd.: *QNX Operating System - System Architekture*, 1996

[16] R. Sedgwick: *Algorithmen in C*++, Addison Wesley, 1992

[17] B. Stroustrup: *The C++ Programming Language*, Third Edition, Addison-Wesley, 1997

[18] J. A. Stankovic: *Real-Time and Embedded Systems*, Computing Surveys 28(1), Seite 205-208, 1996

[19] A. S. Tanenbaum: *Operating Systems - Design and Implementation*, Prentice-Hall International, Inc., 1987

[20] M. Timmerman: *RTOS Market Overview - A follow up*, Real-Time Magazine, 99Q2, 1999

[21] T. Wagner, D. Towsley: *Getting started with POSIX Threads*, Department of Computer Science, University of Massachusetts at Amherst, 1995

[22] M. P. Witzak: *Echtzeitbetriebssysteme - Eine Einführung in Architektur und Programmierung*, Franzis Verlag, 2000

[23] D. Zöbel, W. Albrecht: *Echtzeitsysteme - Grundlagen und Techniken*, International Thomson Publishing, 1995


**LEHRSTUHL FÜR BETRIEBSSYSTEME**

Univ.-Prof. Dr. habil. Thomas Bemmerl

RWTH Aachen, Lehrstuhl für Betriebssysteme

Univ.-Prof. Dr. habil. Thomas Bemmerl

Kopernikusstr. 16

D-52056 Aachen

Germany

Phone: +(49)-241-807634
Fax: +(49)-241-8888339
E-Mail: contact@lfbs.rwth-aachen.de

FTP: ftp.lfbs.rwth-aachen.de
WWW: http://www.lfbs.rwth-aachen.de