Konzepte und funktionaler Vergleich von Thread-Systemen*

Wolfgang Mayerle und Hermann Hellwagner

Institut für Informatik
Technische Universität München

{mayerle,hellwagn}@informatik.tu-muenchen.de

Kurzfassung

Dieses Papier gibt eine allgemeine Einführung in Threads und vergleicht einige derzeit für Arbeitsplatzrechner erhältliche Thread-Systeme. Aufbauend auf einer Motivation und grundlegenden Erläuterung des Thread-Konzepts werden wichtige Aspekte und Probleme von Thread-Bibliotheken vorgestellt. Nach einigen Hinweisen zur Programmierung mit Threads werden mehrere Implementierungen einander gegenübergestellt.

1 Einleitung

Moderne verteilte und parallele Rechensysteme verlangen vielfach auch neue Programmierkonzepte, von denen in diesem Artikel die Programmierung mit *Threads* vertieft behandelt werden soll. Schwerpunkt dieses Beitrags ist ein Vergleich der Funktionen und Konzepte einiger Thread-Systeme auf Arbeitsplatzrechnern (Unix-Workstations und Personal Computer), den die Autoren zur Erhebung des Stands der Technik angestellt haben. Um für den Leser das nötige Verständnis für diesen Vergleich zu schaffen, werden zunächst die wichtigsten Konzepte von Thread-Bibliotheken zusammenfassend dargestellt und einige Hinweise zur Programmierung mit Threads gegeben. Damit ergibt sich insgesamt ein umfassender Überblick zum heutigen Stand der Technik von Thread-Bibliotheken.

Als einführendes Beispiel seien Entwurf und Implementierung eines *effizienten* Datei-Servers betrachtet. Die genaue Arbeitsweise ist dabei für die weiteren Erläuterungen irrelevant. Der Einfachheit halber sei also angenommen, daß in unvorhersehbaren Abständen Aufträge von Client-Programmen zum Lesen oder Schreiben bestimmer Dateiabschnitte eintreffen. Für die Programmierung des Datei-Servers bieten sich prinzipiell mehrere, stark unterschiedliche Ansätze an, die auch Einfluß auf die Effizienz haben:

1. *Ein* Prozeß mit strikt sequentiellem Ablauf:
   Dieses Modell stellt das einfachste dar, denn man kann sich auf synchrone (d.h. blockierende) Systemaufrufe abstützen. Der Nachteil dieses Modells ist die fehlende Parallelität: Während eine Festplatte mit dem Lesen bzw. Beschreiben einer Datei beschäftigt ist, blockiert der Datei-Server. Weitere Anfragen stauen sich in dieser Zeit, obwohl sie möglicherweise beantwortet werden könnten - sei es, weil sie auf eine andere Platte zugreifen wollen oder weil die Operation in einem Datei-Cache durchgeführt werden könnte.
2. Mehrere Prozesse:
   Ein Lösungsansatz für dieses Problem ist auf einem Multitasking-Betriebssystem der Einsatz mehrerer Prozesse mit der gleichen Aufgabe. Die Auftragswarteschlange und der Datei-Cache werden explizit in einem Speicherbereich abgelegt, auf den alle Fileserver-Prozesse zugreifen können (*shared memory*). Mit dieser Methode ist sogar *echte* Parallelität erreichbar: Das Betriebssystem kann auf einem Multiprozessorsystem die Prozesse auf mehrere Prozessoren verteilen. Bei den meisten Arten von Parallelität erkauft man sich jedoch diesen Effizienzgewinn durch mögliche Datenzugriffskonflikte (sog. *races*), die man durch explizite Synchronisationsaufrufe unterbinden muß; der Programmieraufwand ist also zwangsläufig höher als im ersten Modell. Hinzu kommt der Aufwand der Verwaltung des gemeinsamen Speichers. (Insbesondere bei Unix ist *shared memory* ein schlecht integrierter Zusatz zum System). Einen weiteren Nachteil stellt der hohe Ressourcenverbrauch dar, da jeder Prozeß einen eigenen Adreßraum samt Verwaltungsinformationen besitzt. Einen letzten Nachteil bildet die fehlende Kapselung:

* Erschienen in zwei Teilen in: *Praxis der Informationsverarbeitung und Kommunikation,* 3/97 und 4/97

Idealerweise sollte die Parallelität nach außen hin verborgen bleiben, der Datei-Server sollte sich den Clients als ein einziger Prozeß präsentieren.

3. Ein Prozeß mit asynchronen Systemaufrufen:
   Will man die Kapselung erhalten, oder hat man gar nur ein Singletasking-Betriebssystem zur Verfügung, kann man sich wieder auf nur einen Prozeß beschränken und statt blockierenden *asynchrone* Systemaufrufe verwenden (d.h. in unserem Beispiel Aufrufe, die den Auftrag an die Festplatte nur absetzen und sofort wieder zurückkehren). Bei diesem Modell hat man allerdings ein neues Problem. Es können zwei Arten von Meldungen beim Fileserver-Prozeß eintreffen: ein neuer Auftrag oder die Rückmeldung einer Festplatte. Der Datei-Server muß die Rückmeldungen eindeutig zuordnen können, was einen erheblichen zusätzlichen Programmieraufwand darstellt. Ansonsten bietet diese Variante eine gute Kapselung, einen niedrigen Ressourcenverbrauch sowie Quasi-Parallelität. Echt parallele Ausführung ist nicht möglich, da das Betriebssystem nur eine Ausführungseinheit sieht.
4. Mehrere Ausführungseinheiten innerhalb eines Prozesses:
   Will man die Parallelität des Mehrprozeßmodells mit der Kapselung und dem niedrigen Ressourcenverbrauch der Einprozeßmodelle bei möglichst wenig zusätzlichem Programmieraufwand vereinen, kommt man zwangsläufig zu einem System, das dem Programmierer innerhalb eines Adreßraums mehrere Ausführungseinheiten zur Verfügung stellt. Dieses Konzept nennt man *Multithreading*, eine solche Ausführungseinheit heißt *Thread* (oftmals auch als *leichtgewichtiger Prozeß* bezeichnet). Der Prozeß fungiert in diesem System lediglich als Rahmen oder Kontext, der globale Daten gespeichert hat (etwa Codesegment, Datensegment, Dateiinformationen), in dem sich ein oder mehrere Threads bewegen können. Ein Thread benötigt demnach nur noch wenige private Informationen: einen kompletten Satz CPU-Register, einen eigenen Keller sowie weitere Verwaltungsinformationen, von denen einige im Laufe dieses Papiers erklärt werden.

In Tabelle 1 sind die vier Programmiermodelle mit ihren Stärken und Schwächen noch einmal zusammengefaßt. Dabei bedeutet bei „Parallelität“ ein „++“ *echte* (auf Multiprozessoren) und „+“ *Quasi*-Parallelität.

Programmiermodell	1.	2.	3.	4.
Anzahl der Ausführungseinheiten	1	n	1	n
Anzahl der Adreßräume	1	n	1	1
Systemaufrufe	synchron	synchron	asynchron	synchron
Programmieraufwand	++	o	––	+
Parallelität	–	++	+	++
Kapselung	+	–	+	+
Ressourcenverbrauch	+	–	+	+

Tabelle 1 Übersicht Programmiermodelle

Neben den in der Tabelle aufgeführten Vorteilen bieten Threads noch einige weitere:

- Threads sind eine elegante Methode, um aus synchronen Funktionen asynchrone zu machen. Bei der Entwicklung von Bibliotheken kann man sich demnach auf synchrone Funktionen beschränken.
- Threads bieten sehr gute Parallelitätseigenschaften, insbesondere
  - Durchsatzsteigerung durch den Einsatz mehrerer Threads,
  - bessere Antwortzeiten durch asynchrone Abläufe (im Vergleich zu einem strikt sequentiellen Programm) sowie
  - die Möglichkeit echter Parallelität innerhalb eines Prozesses bei Multiprozessorsystemen.
- Viele Programmieraufgaben lassen sich mit dem Thread-Konzept leichter lösen und Programme besser strukturieren als mit herkömmlichen Mitteln.
- Ein Prozeß erhält bei Einsatz von Kernel-Threads (siehe Abschnitt 2.4.1) insgesamt mehr Rechenleistung, was vor allem bei Multiuser-Umgebungen interessant ist.

Einige Anwendungsbereiche, für die sich der Einsatz von Threads lohnt, sind die Realisierung von Servern in Client/Server-Anwendungen, Nutzung von Mehrprozessormaschinen, Programmierung grafischer Bedienoberflächen und Kommunikationsaufgaben [Kleiman et al. 96].

Nach dieser Motivation und Erläuterung des Nutzens von Threads werden im weiteren die wesentlichen Konzepte von Thread-Bibliotheken eingeführt, einige Programmierhinweise zu Threads gegeben sowie abschließend einige der auf Arbeitsplatzrechnern verfügbaren Implementierungen einander gegenübergestellt.

2 Funktionen und Benutzung von Thread-Bibliotheken

In diesem Kapitel werden die gebräuchlichen Schnittstellen von Thread-Bibliotheken vorgestellt (Thread-Management und Synchronisation) sowie einige interne Abläufe erläutert (Scheduling) und Kompatibilitätsprobleme mit herkömmlichen Programmiermethoden angesprochen. Die betrachteten Konzepte und Probleme finden sich in der Mehrzahl der später verglichenen Thread-Implementierungen. Für weitergehende Informationen, Erläuterungen und Beispiele wird vor allem auf [Birrell 91], [Kleiman et al. 96] und [Tanenbaum 92] verwiesen.

2.1 Thread-Management

Alle später betrachteten Systeme haben gemein, daß zu Beginn eines neu erzeugten Prozesses genau ein Thread existiert, der sog. *Initial-Thread*. Ein solcher Prozeß unterscheidet sich zu Beginn also nicht von traditionellen Umgebungen, die nur über schwergewichtige Aktivitätsträger verfügen.

Den Übergang in die Welt des Multithreading gestattet erst ein Satz von Funktionen, die sich mit der Verwaltung von Threads beschäftigen. Tabelle 2 zeigt eine Übersicht der wichtigsten Funktionen.

Name	Beschreibung
`self()`	Liefert die ID des aktuellen Threads
`create()`	Erzeugt einen neuen Thread
`exit()`	Beendet den aktuellen Thread
`join()`	Wartet auf das Ende eines anderen Threads, liefert evtl. dessen Rückgabewert
`detach()`	Löst einen anderen Thread von den übrigen, d.h. kein anderer Thread wird auf ihn warten
`cancel()`	Beendet einen anderen Thread (asynchron)
Abbruchmaskierung	Erlaubt einem Thread Bereiche anzugeben, in denen er ein ihn betreffendes `cancel()` aufschiebt (maskiert)
Abbruchvorsorge	Erlaubt einem Thread Aufräumfunktionen (sog. *cleanup handler*) anzugeben, die im Falle eines unmaskierten `cancel()` noch ausgeführt werden

Tabelle 2 Funktionen zum Thread-Management

Die Funktion `create()` sieht vor, daß beim Aufruf ein Funktionsname als Parameter mit angegeben wird (im Gegensatz zum `fork()` von Unix). Ein Rücksprung aus dieser angegebenen Funktion bewirkt in der Regel ein automatisches `exit()`. (Beim Initial-Thread kann das Verhalten unterschiedlich sein; siehe Abschnitt 2.4.3.) In der Regel ist ein Rückgabeparameter von `create()` die Thread-ID des neu erzeugten Threads.

Die Funktion `join()` ist notwendig, um an Rückgabewerte anderer Threads zu gelangen und „Lücken“ im Speicher (*memory leaks*) zu vermeiden: Nachdem Threads asynchron ablaufen, ist es möglich, daß ein neu erzeugter Thread mit seiner Arbeit früher fertig wird als ein auf ihn wartender. Es ist deshalb die Aufgabe der Thread-Bibliothek, die Rückgabewerte der beendeten Threads bis zum Abruf zwischenzuspeichern.

`detach()` ist das asynchrone Pendant von `join()` und dient lediglich der Vermeidung von *memory leaks*; hier kann verständlicherweise kein Rückgabewert geliefert werden.

`cancel()` erlaubt das asynchrone Beenden eines Threads, was ein neues Problem aufwirft: Besitzt der Ziel-Thread gerade exklusive Ressourcen (z.B. Speicher, Synchronisationsmittel), so werden diese nicht automatisch freigegeben. Es bieten sich zwei Möglichkeiten an, diesem Problem zu begegnen: nur an bestimmten Programmstellen ein `cancel()` zu erlauben (Abbruchmaskierung) oder Vorkehrungen für den `cancel()`-Fall im voraus zu treffen (Abbruchvorsorge). Die Abbruchvorsorge ist üblicherweise durch einen Funktionskeller realisiert, auf den man sog. *cleanup handler* explizit speichern (*push*) und explizit entfernen und ggf. ausführen (*pop*) kann. Bei einem unmaskierten `cancel()` werden alle Funktionen auf dem Stapel implizit ausgeführt.

2.2 Synchronisation

Um die Abläufe und Zugriffe auf gemeinsame Daten von mehreren (quasi-)parallel ablaufenden Threads koordinieren zu können, werden Möglichkeiten zur (Interthread-)Synchronisation benötigt. (Unabhängig davon gestatten viele Thread-Implementierungen die Benutzung eben dieser Mechanismen über Prozeßgrenzen hinweg. Hier sollen jedoch in erster Linie die prozeßlokalen Synchronisationsmittel behandelt werden.)

Generell kann man feststellen, daß es zu jedem Synchronisationsobjekt ein Funktionspaar für dessen Initialisierung und Beendigung gibt. Das hat *nichts* mit der Speicherverwaltung dieser Objekte zu tun. Es gibt lediglich dem zugrundeliegenden Thread-System die Möglichkeit, diese Objekte in seine internen Datenstrukturen einzubinden und zu initialisieren bzw. daraus zu entfernen.

2.2.1 Monitore

Ein Monitor im Sinne von [Hoare 74] stellt ein Synchronisationsmittel für ein von mehreren Threads verwendetes Datenobjekt dar. Ein Monitor besteht aus

- dem zu schützenden Datenobjekt,
- einem *Mutex* (*mutual exclusion lock*, einem Mittel zum wechselseitigen Ausschluß) und
- einer endlichen Anzahl von Bedingungsvariablen (*condition variables*).

Mutexes

In Tabelle 3 sind die verschiedenen Funktionen für Mutexes aufgeführt. Die Thread-Bibliothek garantiert dem Programmierer, daß zwischen den mit (a) und (b) gekennzeichneten Teiloperationen einer `mutex_lock(m)`-Operation von keinem anderen Thread ein `mutex_lock(m)` erfolgreich ausgeführt werden kann. Diese Forderung ist notwendig, damit nicht mehrere Threads in den Besitz desselben Mutexes gelangen können.

Name	Beschreibung
`mutex_init()`	Initialisiert den Mutex
`mutex_destroy()`	Beendet den Mutex
`mutex_lock()`	Blockiert solange, bis der Mutex frei ist (a) und nimmt ihn dann in Besitz (b)
`mutex_unlock()`	Gibt den Mutex frei (nur der Besitzer kann das)

Tabelle 3 Funktionen für Mutexes

Implementationsunterschiede gibt es hier in der exakten Funktionsweise von `mutex_lock()` und `mutex_unlock()`. Aufschluß über die Semantik liefert ein abermaliges Anfordern eines Mutexes durch seinen Besitzer ohne vorherige Freigabe. Es kann einer der folgenden drei Fälle eintreten:

- Verklemmende Mutexes:
  Der Thread blockiert und kann nicht mehr fortgeführt werden, weil er auf die Freigabe des Mutex wartet, welche er jedoch nur selbst veranlassen könnte. (Zu Verklemmungen (*deadlocks*) allgemein siehe Abschnitt 3.2.5.)
- Nicht-verklemmende, nicht-rekursive Mutexes:
  Die Thread-Bibliothek erkennt die drohende Verklemmung und liefert eine Fehlermeldung.
- Rekursive Mutexes:
  Im Gegensatz zu beiden obigen Varianten wird bei dieser Variante keine Fehlermeldung zurückgegeben. Die Bibliothek zählt intern die Zahl der verschachtelten `mutex_lock()`-Aufrufe und gibt erst nach einer ebenso großen Anzahl von `mutex_unlock()`-Aufrufen den Mutex wieder frei. Es sei noch einmal betont, daß hier nur der *Besitzer* einen Mutex nochmals erhalten kann; Vergabe an einen anderen Thread würde das Prinzip des wechselseitigen Ausschlusses verletzen.

Es lassen sich für alle drei Ansätze Szenarien finden, die ihre Existenz rechtfertigen. Jedoch werden in den meisten Fällen die rekursiven Mutexes die angenehmsten sein; der Programmierer kann problemlos und ohne weiteren Programmieraufwand aus einer mit einem Mutex geschützen Routine eine weitere, ebenfalls mit demselben Mutex geschützte Routine aufrufen.

Bedingungsvariable und Monitore

Mutexes bilden zusammen mit Bedingungsvariablen, deren Funktionen in Tabelle 4 aufgeführt sind, sog. *Monitore*. Die Bausteine eines Monitors haben dabei folgende Funktionen: Der Mutex stellt die Serialisierung für das zu schützende Datenobjekt sicher, die Bedingungsvariablen erlauben es dem Programmierer, in die Ausführungsreihenfolge der Threads gezielt einzugreifen.

Name	Beschreibung
`cond_init()`	Initialisiert eine Bedingungsvariable
`cond_destroy()`	Beendet eine Bedingungsvariable
`cond_wait()`	Der aufrufende Thread gibt den angegebenen Mutex frei (a), blockiert solange (b), bis er über die spezifizierte Bedingungsvariable benachrichtigt wird, und nimmt den Mutex dann wieder in Besitz
`cond_signal()`	Benachrichtigt einen auf die spezifizierte Bedingungsvariable wartenden Thread, sofern es einen gibt
`cond_broadcast()`	Benachrichtigt alle auf die spezifizierte Bedingungsvariable wartenden Threads

Tabelle 4 Funktionen für Bedingungsvariablen

Ähnlich wie bei `mutex_lock()` muß die Thread-Bibliothek garantieren, daß zwischen den mit (a) und (b) gekennzeichneten Teiloperationen von `cond_wait(c,m)` kein `cond_signal(c)` bzw. `cond_broadcast(c)` ausgeführt wird. Bei Verletzung dieser Bedingung würde eine eventuelle Benachrichtigung ungehört verhallen, und der zu Beginn unterbrochene Thread würde möglicherweise ewig auf das Eintreten der gewünschten Bedingung warten. Bedingungsvariable haben *kein* Gedächtnis: Wartet zum Zeitpunkt der Benachrichtigung kein Thread auf eine Variable, hat die Benachrichtigung keine Wirkung. Dieses Phänomen ist unter dem Namen *lost-wakeup problem* bekannt.

Eine wichtige Eigenschaft der Funktion `cond_wait()`, die im Zusammenhang mit der Benutzung von Threads (Abschnitt 3) noch zur Sprache kommen wird, ist die Möglichkeit, daß sie intern zweimal blockieren kann: einmal, um auf die Variable zu warten, und zum zweiten Mal, um den Mutex zu bekommen.

Bild 1 veranschaulicht die Wirkungsweise eines Monitors sowie das Zusammenspiel von Mutex und Bedingungsvariablen. Der Programmablauf bei den einzelnen Funktionen ist von oben nach unten laufend zu verstehen. Umrahmte Einheiten müssen atomar ausgeführt werden, ansonsten ist der jeweils nebenstehende Fehler nicht auszuschließen.

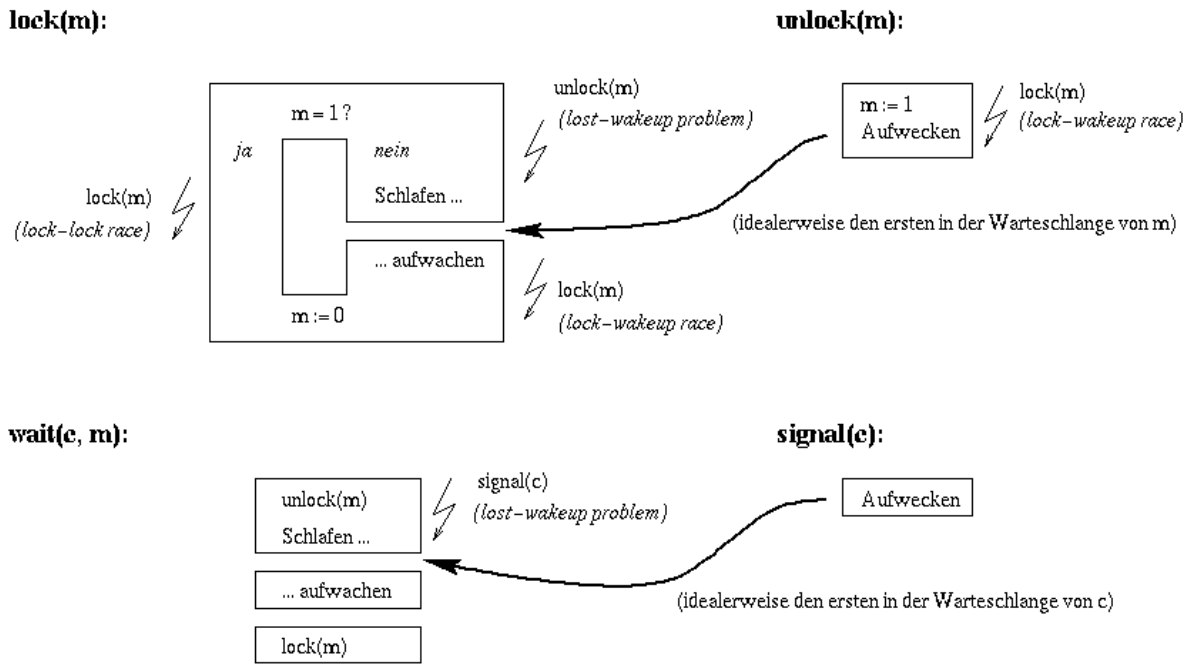

Bild 1 Monitore

Beispiel

Ein kleines Beispielprogramm soll die Funktionsweise von Monitoren illustrieren. Es handelt sich um ein sog. *Shared/Exclusive-Lock* (auch unter dem Namen *Multiple Readers/Single Writer-Lock* bekannt), welches seinerseits ein etwas komplexeres Synchronisationsmittel darstellt (Bild 2).

Die drei dafür erforderlichen Funktionen sind jeweils in einen `mutex_lock()-mutex_unlock()-` Rahmen gebettet, was zur Folge hat, daß zu einem Zeitpunkt *höchstens* ein Thread in einer der drei Funktionen *aktiv* ist. Das hier implementierte Protokoll ist einfach:

- Ein Leser-Anwärter bekommt die Erlaubnis, falls es im Moment keinen Schreiber gibt; andernfalls muß er warten.
- Ein Schreiber-Anwärter bekommt die Erlaubnis, falls es im Moment weder Leser noch Schreiber gibt; andernfalls muß er warten.
- Gibt ein Thread seine Rechte auf und ist kein anderer mehr aktiv, werden alle wartenden Threads geweckt.

```
mutex_t mutex;
cond_t cond;
int count;   /* Anzahl der Leser oder -1, falls Schreiber aktiv */

void rw_rdlock(void)
{
  mutex_lock(mutex);
  while (count < 0)          /* Wartet nur, wenn gerade */
    cond_wait(cond, mutex); /* ein Schreiber aktiv ist */
  count++;
  mutex_unlock(mutex);
}

void rw_wrlock(void)
{
  mutex_lock(mutex);
  while (count != 0)         /* Wartet, bis alle fertig sind */
    cond_wait(cond, mutex);
  count = -1;
  mutex_unlock(mutex);
}

void rw_unlock(void)
{
  mutex_lock(mutex);
  if (count < 0)
    count = 0;               /* unlock() eines Schreibers */
  else
    count--;                 /* unlock() eines Lesers */
  if (count == 0)            /* Alle werden geweckt */
    cond_broadcast(cond);
  mutex_unlock(mutex);
}
```

Bild 2 Eine *Shared/Exclusive-Lock*-Implementierung mit Monitoren

2.2.2 Ereignisvariable

Monitore stellen ein einfaches, flexibles und gut durchschaubares Synchronisationsmodell zur Verfügung, von dem sämtliche anderen Synchronisationsmittel abgeleitet werden können. Ein weiterer verbreiteter Mechanismus sind Ereignisvariablen (*event variables*).

Da die Ereignisvariablen - im Gegensatz zu den Monitoren - nicht auf ein wohldefiniertes Konzept zurückgehen, haben sie stark implementierungsabhängigen Funktionsumfang. Sie sind einerseits stärker eingeschränkt als Bedingungsvariable, weil sie kein Zusammenspiel mit Mutexes bieten, andererseits aber mächtiger, weil sie ein Gedächtnis besitzen.

Man unterscheidet im allgemeinen zwischen
- *manuell* rückzusetzenden Ereignisvariablen und
- *automatisch* rückgesetzten Ereignisvariablen.

Die gebräuchlichen Funktionen für Ereignisvariablen sind in Tabelle 5 zusammengestellt. Daraus wird u.a. klar, daß bei manuell rückzusetzenden Ereignisvariablen das Ereignis nur durch ein explizites `event_reset()` zurückgesetzt wird, wohingegen bei automatischen zusätzlich bei erfolgreicher Rückkehr aus einem `event_wait()`-Aufruf das Ereignis gelöscht wird.

Die Unteilbarkeit der Operationen (a) und (b) bei `event_wait()` machen automatisch rückgesetzte Ereignisvariable zum Gegenstück von `cond_signal()`. Es wird nur ein Thread geweckt, auch wenn mehrere warten. Manuelle rückzusetzende Ereignisvariable hingegen entsprechen `cond_broadcast()`, wenn man von der noch ausstehenden manuellen Rücksetzung absieht. Bei beiden Analogien ist noch zu berücksichtigen, daß Ereignisvariable ein Gedächtnis besitzen, Bedingungsvariable nicht.

Name	Beschreibung
`event_init()`	Initialisiert eine Ereignisvariable und legt deren Typ fest (automatisch oder manuell zurückgesetzt)
`event_destroy()`	Beendet eine Ereignisvariable
`event_wait()`	Blockiert solange, bis Ereignis gesetzt ist (a) und setzt dann im Fall einer automatischen Ereignisvariablen das Ereignis zurück (b)
`event_set()`	Setzt das Ereignis
`event_reset()`	Setzt das Ereignis zurück

Tabelle 5 Funktionen für Ereignisvariablen

2.2.3 Weitere Synchronisationsmittel

Neben den Monitoren und Ereignisvariablen gibt es noch zwei andere verbreitete Synchronisationsmittel, die jedoch nur eingeschränkte Flexibilität bieten:
- Semaphore:
  Allgemeine Semaphore sind Verfügbarkeitszähler, binäre Semaphore sind ein Spezialfall und sind identisch mit Mutexes ohne Besitzereigenschaft.
- Kritische Bereiche (*critical sections*):
  Diese werden bei manchen Systemen als schnellere (da ausschließlich prozeßlokale) Alternative zu den Mutexes angeboten. Es stellt sich dem Programmierer jedoch die Frage, weshalb dazu eigene Systemaufrufe notwendig sind.

2.2.4 Zusammenfassung

An dieser Stelle sollen zur Vertiefung des Verständnisses schematisch die Zusammenhänge der verschiedenen vorgestellten Synchonisationsmittel aufgezeigt werden (Bild 3). Es zeigt sich, daß die Unterschiede konzeptionell zum Teil nicht groß sind; sie sind es aber in der Benutzung der Synchronisationsmechanismen.

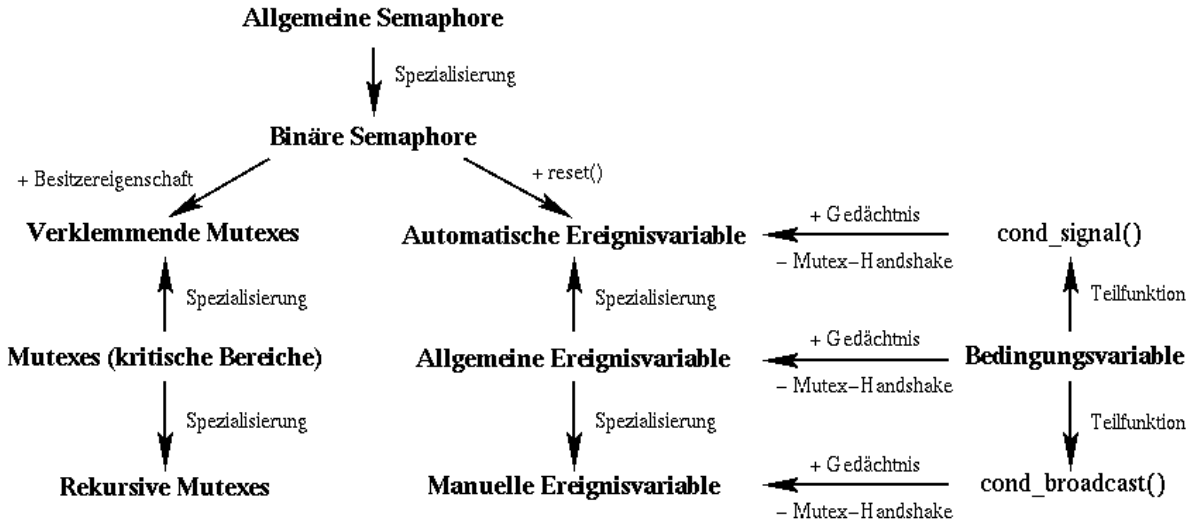

Bild 3 Schematischer Überblick der Synchronisationsmittel

2.3 Scheduling

Unter Berücksichtigung der Synchronisationsfunktionen kann ein Thread im wesentlichen einen von zwei Zuständen einnehmen: *wartend* (blockiert) oder *lauffähig*.

Unter Scheduling versteht man die Zuteilung der lauffähigen Threads (*nicht* der Prozesse) an die vorhandenen Prozessoren durch den *Scheduler*, welcher einen Teil der Thread-Bibliothek darstellt. Das Umstellen von „lauffähig“ auf tatsächlich „laufend“ nennt man *context switch*. Zu einem Zeitpunkt kann es höchstens so viele laufende Threads wie Prozessoren geben.

Präemptive Scheduler sind in der Lage, gerade laufende Threads zu unterbrechen. *Kooperative* Scheduler tun dies im Gegensatz dazu nicht; sie werden heute kaum noch eingesetzt.

2.3.1 Prioritäts-Scheduling

Am verbreitetsten ist das sog. *Prioritäts-Scheduling*, bei dem jedem Thread eine diskrete Priorität zugeordnet wird. Der Scheduler sorgt dafür, daß zu jedem Zeitpunkt die Prozessoren mit den lauffähigen Threads der höchsten Prioritäten belegt sind. Dabei kann es zu einem Streitfall kommen, in dem mehr Threads derselben Priorität vorhanden sind als Prozessoren. Hier bieten sich zwei verschiedene Strategien an:

- *First-In First-Out / Run-to-Completion*:
  Die ersten der in Frage kommenden, wartenden Threads in der Warteliste bekommen die verfügbaren Prozessoren.
- *Round-Robin / Timeslicing*:
  Diese Threads teilen sich die verfügbaren Prozessoren per Zeitmultiplexing.

Darüber hinaus unterscheidet man zwei Klassen von Prioritäts-Scheduling:

- Statisch (feste Priorität):
  Die Priotität eines Threads wird *nicht* automatisch vom System verändert.
- Dynamisch (veränderliche Priorität):
  Die Priorität eines Threads *wird* in einem gewissen Rahmen automatisch vom System verändert. Typischerweise wird ein Thread befördert, wenn das Ereignis eintritt, auf das er gewartet hat, und er wird heruntergestuft, wenn er seine Zeitscheibe vollständig aufgebraucht hat. Diese Variante ist zwangsläufig immer mit dem Round-Robin-Verfahren verbunden.

In vielen Systemen findet man eine Mischung dieser zwei Klassen. Dabei findet die statische Variante insbesondere bei Echtzeitanforderungen Anwendung, die dynamische bei sonstigen Benutzerprozessen. Generell werden bei solchen Systemen Threads mit fester Priorität denen mit dynamischer bevorzugt. Manche Scheduler variieren zusätzlich je nach Systemauslastung die Länge der Zeitscheibe.

2.3.2 Besondere Protokolle

Das Scheduling mit fester Priorität hat zwei gravierende Nachteile:

- Systemstillstand:
  Falls es einem Benutzer gelingt, so viele Threads mit der höchsten Priorität zu erschaffen, wie es Prozessoren gibt, und diese Threads nie in einen Wartezustand übergehen (z.B. Endlosschleife), ist ein Systemstillstand erreicht.
- Problem der *Prioritätsinversion*:
  Darunter versteht man das Phänomen, daß ein niedrigpriorer (also langsamer) Thread einen Mutex besitzt, den ein hochpriorer Thread anfordert. Da der Besitzer niedrige Priorität hat, ist es gut möglich, daß sich dieser Zustand geraume Zeit hält; man spricht von *priority inversion*.

Als Lösung für das erste Problem kommen zum einen Prioritätsbeschränkungen für die Benutzer und zum anderen eine fairere Variante in Betracht, die etwa abhängig von der Priorität den jeweiligen Threads eine unterschiedliche Zeitscheibe zuteilt. Die letztere Methode gewährleistet sogar, daß kein Thread verhungert. Lediglich eines der später vorgestellten Systeme bietet die Möglichkeit eines fairen Scheduling.

Dem Problem der Prioritätsinversion kann man u.a. durch ein *Prioritätsvererbungsprotokoll* begegnen (*priority inheritance protocol*). Dabei erbt der Besitzer eines Mutex vorübergehend die höchste Priorität aller auf den Mutex wartenden Threads. Erstaunlicherweise hat keines der getesteten Systeme dieses Protokoll implementiert.

2.4 Kompatibilitätsprobleme

In diesem Abschnitt werden die Aspekte einer Thread-Bibliothek angesprochen, die vorwiegend aus Kompatibilität zu traditionellen Systemen resultieren.

2.4.1 User- oder Kernel-Threads

Eine prinzipielle Entscheidung, die man bei der Entwicklung einer Thread-Bibliothek zu treffen hat, betrifft deren Integration in das Gesamtsystem. Man kann zum einen das Threadkonzept in den Betriebssystemkern einbetten (*kernel threads*), oder man kann Threads als eine eigenständige Zwischenschicht zwischen Betriebssystem und Anwenderprogrammen zur Verfügung stellen (*user threads*).

Der Unterschied zwischen beiden Ansätzen ist lediglich die Kenntnis bzw. Unkenntnis des Betriebssystems von Threads, woraus sich jedoch mehrere Konsequenzen ergeben. Diese Konsequenzen schlagen sich in Form von Vor- und Nachteilen für beide Varianten nieder, wovon die gravierendsten aus der Sicht der User-Threads im folgenden kurz aufgeführt werden. (Kernel-Threads haben genau die Umkehrung der Vor- und Nachteile der User-Threads.)

Vorteile der User-Threads sind:

- Sie sind schnell zu implementieren.
- Sie sind leicht auf ein anderes System zu portieren.
- Sie belasten das Betriebssystem wenig, weil z.B. keine Betriebssystem-Verwaltungsdaten oder Betriebssystemaufrufe erforderlich sind; folglich sind Kontextwechsel und Synchronisation effizient.

Nachteile von User-Threads sind:

- Innerhalb eines Prozesses ist keine echte Parallelität möglich.
- Blockierende Systemaufrufe machen entweder Parallelität zunichte (beim Ignorieren des Problems) oder führen zu Effizienzeinbußen (durch explizite Tests vor den Aufrufen, ob diese denn zu Blockierungen führen würden).

Beide Nachteile rühren daher, daß zwei nicht miteinander kommunizierende Scheduler tätig sind: der Prozeß-Scheduler im Betriebssystemkern zum einen und der Thread-Scheduler auf Ebene des Benutzerprogramms auf der anderen Seite.

Bei einigen ausgefeilten Systemen findet man eine Kombination von User- und Kernel-Threads, was dem Programmierer die Vorteile beider Systeme bei wenig höherem Programmieraufwand einbringt. In solchen Systemen gibt es dann zwei Scheduler, die ihre Tätigkeit aufeinander abstimmen können.

2.4.2 Thread-sichere und wiedereintrittsfeste Bibliotheken

Viele bestehende Bibliotheken sind für Ausführung durch Threads ungeeignet und liefern unvorhersehbare Ergebnisse, wenn sie in einer Funktion vom Scheduler unterbrochen und zur Ausführung einer anderen Funktion (für einen anderen Thread) gezwungen werden. Der Hauptgrund liegt im Einsatz statischer Variablen in den Bibliotheken.

Thread-sichere (*thread-safe*) Funktionen bzw. Bibliotheken können von mehreren Threads gleichzeitig ausgeführt werden, ohne daß der Benutzer dafür spezielle Vorkehrungen treffen muß. Die erforderlichen Synchronisationen, z.B. beim Zugriff auf Statusvariablen in der Bibliothek, haben die Threads intern vorzunehmen. Wiedereintrittsfeste (*re-entrant*) Bibliotheken können über die übliche Benutzung durch Threads hinaus auch noch von anderen Mechanismen, z.B. Signal-Handlern, nebenläufig betreten werden [Kleiman et al. 96].

Um Bibliotheken zumindest thread-sicher zu machen, bieten sich zwei Alternativen an:
- *Jacketing*:
  Die gesamte Bibliothek oder die ungeeigneten Funktionen werden (z.B. mittels *Wrapper*-Funktionen) unter gegenseitigen Ausschluß gestellt. Die Durchführung ist verhältnismäßig einfach, jedoch erhält der Benutzer der Bibliothek schlechte Parallelität.
- Neuentwicklung:
  Die Bibliothek ist neu zu schreiben, unter Berücksichtigung der Multithreading-Ausführungsumgebung. Dieser Ansatz steht im Gegensatz zum ersten, da der Bibliotheksentwickler u.U. erheblichen Programmieraufwand betreiben muß. Andererseits kann der Benutzer von der gewonnenen hohen Parallelität profitieren.

2.4.3 Prozeßende

Auch was das Ende eines Prozesses betrifft, gibt es in gängigen Thread-Bibliotheken mindestens zwei verschiedene Varianten:
- Bei vielen Systemen ist ein Prozeß beendet, wenn der Initial-Thread beendet wird. Dies wirft u.a. die Frage auf, was mit den anderen Threads geschieht. Es scheint hier durchgehend so zu sein, daß die restlichen aktiven Threads beendet und deren evtl. vorhandene Cleanup-Handler kurzerhand ignoriert werden.
- Bei einigen anderen Systemen hingegen wird der Prozeß mit dem Ende des letzten Threads beendet, der nicht zwangsläufig der Initial-Thread sein muß.

Sofern es in der jeweiligen Thread-Umgebung die Standard-C-Funktion `exit()` gibt, bewirkt deren Aufruf generell ein (abnormales) Prozeßende. Es ist in diesen Fällen zu beachten, daß ein Rücksprung aus `main()` einen impliziten Aufruf von `exit()` zur Folge hat.

2.4.4 Ausnahmebehandlung

Bei der Einführung von Multithreading in ein bestehendes System stellt sich das Problem, wie man die bisherige Ausnahmebehandlung (*signal handling* in Unix-Terminologie) übernimmt bzw. anpaßt.

Man kann folgende Zustellungsvarianten für Ausnahmen (*exceptions*) unterscheiden:
- Zustellung an einen Ziel-Thread:
  Eine Ausnahme ist eindeutig mit einem Ziel-Thread verbunden. Bei manchen Systemen ist das sogar immer garantiert.
- Gemischte Zustellung:
  Falls ein solcher Thread existiert, wird wie in der ersten Variante verfahren; ansonsten wird die Ausnahme einem Thread zugestellt, der auf Ausnahmen wartet. Gibt es diesen nicht, erhält sie ein beliebiger Thread.

In der Regel wird eine Ausnahme nur einem Thread zugestellt. Es gibt jedoch Systeme, bei denen z.B. das Prozeßendesignal allen Threads zugestellt wird (OS/2).

Bei manchen Systemen gibt es eine Möglichkeit, Ausnahmen per Thread zu maskieren, d.h. (vorübergehend) aufzuschieben, sowie die Möglichkeit, pro Thread einen Ausnahme-Handler bereitzustellen.

3 Programmierhinweise

Dieses Kapitel gibt einige wichtige Hinweise für die Benutzung von Threads. Konkret werden verschiedene Organisationsmodelle sowie einige Aspekte der Korrektheit und der Effizienz betrachtet. Dieses Themas wird sehr anschaulich in [Birrell 91] behandelt.

3.1 Verschiedene Organisationsmodelle

Threads können sich untereinander die Arbeit auf verschiedene Weise aufteilen:

- Team (Datenaufteilung):
  Alle Threads sind gleichberechtigt und führen die gleiche Arbeit aus.
- Dispatcher / Worker:
  Es gibt einen ausgezeichneten Koordinations-Thread, der die zu erledigenden Aufgaben an die Arbeiter-Threads verteilt.
- Pipeline (Codeaufteilung):
  Jeder Thread führt eine andere logische Einheit der gesamten Arbeit aus.

Die Pipeline-Methode hat gegenüber den anderen den Nachteil, daß sie *nicht skalierbar* ist, weil die Anzahl der Pipeline-Stufen zur Zeit des Programmierens i.a. festgelegt werden muß.

3.2 Zur Korrektheit

In diesem Abschnitt werden einige mehr oder weniger subtile Fehlerquellen aufgezeigt, die beim Programmieren in einer Multithreading-Umgebung zu beachten sind.

3.2.1 Mutex als Invariante

In vielen Fällen ist es hilfreich, sich einen Mutex nicht nur als Sequentialisierungsmechanismus vorzustellen, sondern sich dessen bewußt zu sein, daß er ein Garant für die Konsistenz seines Datenobjekts ist: Falls der Mutex nicht belegt ist, befindet sich sein zu schützendes Objekt in einem konsistenten Zustand. Dieser Zustand sollte in Verbindung mit dem Mutex dokumentiert sein.

Die Kenntnis der Invarianten stellt oft eine große Erleichterung für das Verständnis des Programms dar. Als Folge daraus läßt sich auch die Korrektheit leichter beurteilen.

3.2.2 Prüfung von Bedingungen in Monitoren mittels `while` statt `if`

Über Bedingungsvariablen werden nur Nachrichten mitgeteilt, sie speichern keinen Zustand. Deshalb benötigt man noch eine prüfbare (Boolesche) Bedingung, die anzeigt, ob im Moment die gewünschte Situation tatsächlich vorliegt.

Man ist also versucht, folgendes Codegerüst zu benutzen:

```
lock(mutex);
if (!bedingung)
  cond_wait(cond_var, mutex);
... /* Arbeit */
unlock(mutex);
```

Diese Methode berücksichtigt jedoch folgendes nicht: Nach der Rückkehr aus `cond_wait()` muß die Bedingung *nicht* zwangsläufig erfüllt sein. Dies hat zwei Gründe:

- Durch das zweimalige interne Blockieren innerhalb von `cond_wait()` ist nicht auszuschließen, daß dazwischen ein anderer Thread den Mutex bekommt und das Datenobjekt verändert. Die urspünglichen Monitore nach Hoare garantieren die Gültigkeit, indem `cond_wait()` nur einmal blockiert. Man ist dadurch jedoch in den Möglichkeiten der Effizienzsteigerung stärker eingeschränkt.
- Bei einigen echt parallelen Monitor-Implementierungen wurde die Forderung gelockert, daß bei einem `cond_signal()` höchstens ein Thread geweckt wird, um den Implementiereren die Arbeit ein wenig zu erleichtern. Es können in seltenen Fällen also mehrere Threads geweckt werden.

Aus diesen Gründen sollte man das `if` in obigem Codeausschnitt in ein `while` umwandeln; das Programm wird nicht langsamer, sondern nur klarer:

```
while (!bedingung)
  cond_wait(cond_var, mutex);
```

Die *Shared/Exclusive-Lock*-Implementierung in Bild 2 benutzt bereits diese sichere Methode.

3.2.3 Thread-sicherer Code

Um eine Funktion thread-sicher oder wiedereintrittsfest, d.h. geeignet für eine Multithreading-Umgebung zu machen, ist es sehr ratsam, auf globale Daten zu verzichten. Das hat meist zur Folge, daß die Schnittstelle um einige Parameter erweitert werden muß, welche von nun ab vom Aufrufer mit zur Verfügung gestellt werden müssen.

Manche Bibliotheken bieten Funktionen zur Unterstützung sog. thread-spezifischer Daten (*thread specific data*) an, mittels derer dennoch eine Art statischer Daten pro Ausführungseinheit zur Verfügung gestellt werden kann. Die Variable `errno` insbesondere ist in allen Umgebungen als ein solcher Datentyp vorhanden.

3.2.4 Zugriffskonflikte

Unter Zugriffskonflikten (*data races*) versteht man den konkurrierenden und unsynchronisierten Zugriff mehrerer Ausführungseinheiten auf ein Datenobjekt. Um korrekte Nutzung und Modifikationen der gemeinsamen Datenobjekte sicherzustellen, muß der Programmierer die vorhin vorgestellten Synchronisationsmittel einsetzen.

3.2.5 Verklemmungen

Durch den Einsatz von Synchronisationsmitteln werden die Races nicht eliminiert, sondern streng genommen nur zu den Synchronisationsmitteln verschoben; man spricht hier i.a. jedoch nicht mehr von Races.

Werden Synchronisationsmittel falsch eingesetzt, können sich aus deren Races sog. Verklemmungen (*deadlocks*) entwickeln. Verklemmungen sind zyklische Wartebedingungen, die zur Folge haben, daß weiterer Forschritt im Programmablauf nicht mehr erzielt werden kann. Sie können bei Monitoren auf zweierlei Arten auftreten (siehe auch Bild 4):

- Ressourcen werden in unterschiedlicher Reihenfolge angefordert.
- Das Programm verzweigt in eine logisch tiefer liegende Ebene, ohne die derzeit belegten Ressourcen freizugeben (*nested monitor problem*).

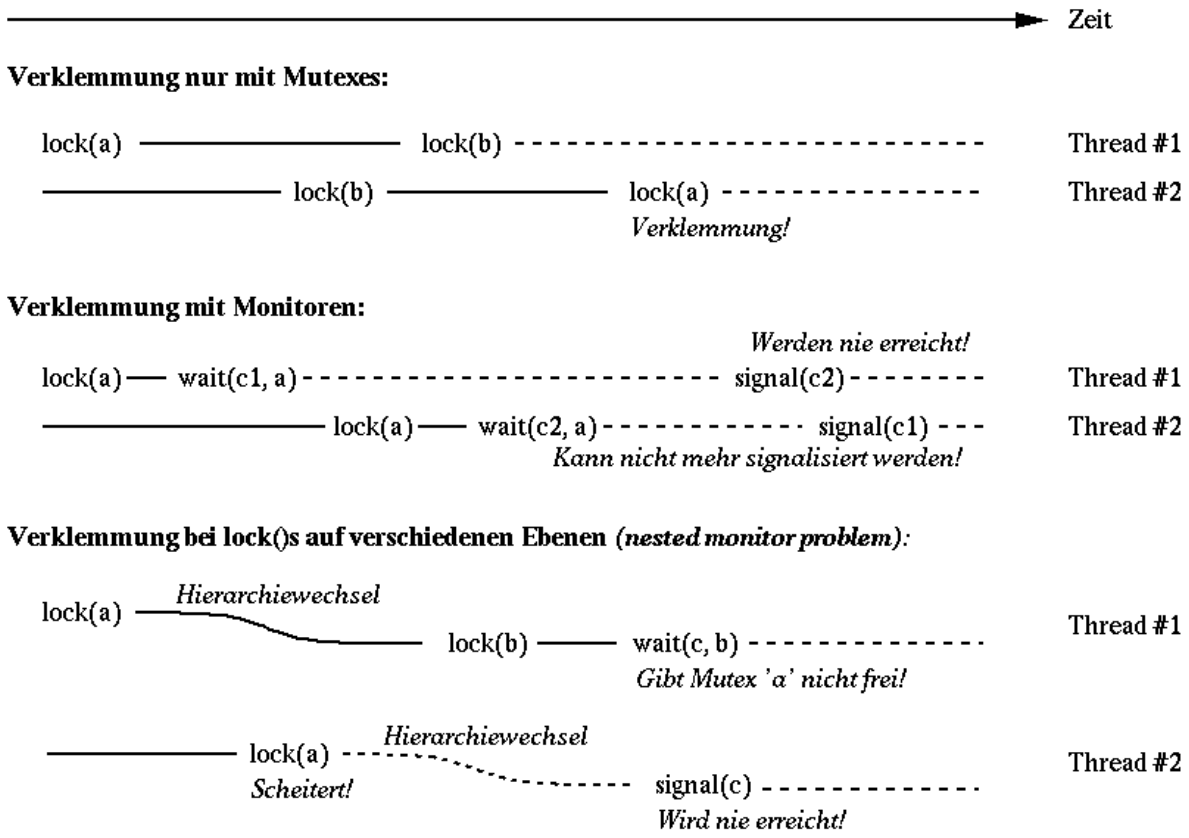

Bild 4 Verklemmungen

Um die erste Fehlerquelle zu vermeiden, muß man auf den Synchronisationsobjekten eine *partielle Ordnung* festlegen, welche die Reihenfolge bestimmt, in der man die Betriebsmittel anfordern darf.

Für das zweite Problem gibt es ebenfalls keine befriedigende allgemeine Lösung. Es bleibt nur der Appell an die Disziplin des Programmierers, entweder beim Besitz von Ressourcen *nicht* die logische Ebene zu wechseln, oder, falls dies docherforderlich ist, dann die Ressourcen vorher freizugeben.

Bei genauerer Betrachtung von Bild 4 bemerkt man, daß Verklemmungen in Verbindung mit Bedingungsvariablen (Fälle 2 und 3) fast schon zwangsläufig auftreten: Wenn in diesen Fällen ein `cond_wait()`-Aufruf stattfindet, ergibt sich auch tatsächlich eine Verklemmung. Das ist im Vergleich zu den schwer reproduzierbaren Verklemmungen bei *lock races* ein erheblicher Vorteil.

3.2.6 Unfairness

Sobald man selbst mittels Bedingungs- oder Ereignisvariablen ins Scheduling eingreift, muß man sich überlegen, ob alle Threads auch fair behandelt werden. Die naheliegendste Lösung für das *Shared/Exclusive-Locking* (siehe Bild 2) ist hierfür ein gutes Beispiel. Es kann bei einem stark belasteten System nämlich durchaus der Fall eintreten, daß es zu jedem Zeitpunkt mindestens einen aktiven Leser gibt. Das hat zur Folge, daß kein Schreiber mehr an die Reihe kommt: Die Schreiber verhungern (*starvation*).

Abhilfe schaffen hier nur kompliziertere und meist auch fehleranfälligere Ablaufprotokolle. Man sollte sich im Einzelfall im voraus überlegen, ob der Aufwand für faire Protokolle gerechtfertigt ist.

Die Wartebedingungen bei allen blockierenden Synchronisationsfunktionen (wie etwa `mutex_lock()` oder `cond_wait()`) sind typischerweise durch eine FIFO-Warteschlange, die nach Prioritäten sortiert ist, realisiert. Dieser Umstand kann sich ebenfalls auf die Fairneß des Programms auswirken.

3.3 Zur Effizienz

3.3.1 Geschäftiges Warten

Geschäftiges Warten (*busy waiting*) auf das Eintreten einer Bedingung ist in den meisten Fällen zu vermeiden. Der Grund ist offensichtlich: Eine Ausführungseinheit, die nur auf das Eintreten eines Ereignisses wartet, kann im Moment keine sinnvolle Arbeit erledigen und sollte demnach auch keine Rechenzeit verbrauchen. Benutzt man hierzu keine Synchronisationsmittel, sondern eine Warteschleife, hat das Betriebssystem keine Möglichkeit zu erkennen, ob ein Thread gerade wartet oder arbeitet. Demnach wird ihm in vielen Fällen unnötig Rechenzeit zugeteilt.

Außerdem zieht geschäftiges Warten meist auch noch ein Korrektheitsproblem nach sich, weil nach Beendigung der Warteschleife die Bedingung schon wieder verletzt sein könnte. Es fehlt hier die Atomizität des Wartens und Belegens, welche man bei Synchronisationsmitteln hat.

3.3.2 Feingranulares Sperren

Die sequentiellen Anteile eines Programms sollten möglichst kurz sein. Exklusive Ressourcen sollten daher für einen möglichst kurzen Zeitraum gesperrt sein (mittels Mutexes oder Semaphoren).

Dies zieht leider wieder einen Konflikt nach sich: Feingranulares Sperren erlaubt zwar höhere Parallelität, erfordert aber auch mehr Programmieraufwand, macht das Programm i.a. schlechter lesbar und erhöht die Gefahr von Verklemmungssituationen.

3.3.3 Reduktion von Kontextwechseln

Falls sehr viel Synchronisationsaufwand getrieben wird, kann man durch Vermeidung von überflüssigen Kontextwechseln noch etwas an Effizienz gewinnen.

Bei Monitoren führt das zweimalige Blockieren innerhalb eines `cond_wait()` sehr schnell zu überflüssigen Kontextwechseln. Durch etwas zusätzlichen Programmieraufwand kann man deren Anzahl jedoch reduzieren.

So ist es oft möglich, ein `cond_signal()`, das kurz vor Ende des Monitors steht, *hinter* den `mutex_unlock()`-Aufruf zu verschieben. Das *lost-wakeup problem* kann in diesem Fall nicht auftreten, da zum Zeitpunkt des `mutex_unlock()` die zu signalisierende Bedingung schon gültig ist; ein sich danach erkundigender Thread geht also gar nicht in ein `cond_wait()`.

Gerade bei Multiprozessoren ist von der Verwendung des `cond_broadcast()` abzuraten, da eine Reihe unnötiger Kontextwechsel stattfände: Die meisten Threads würden zwar geweckt, aber gleich wieder beim Versuch scheitern, den Mutex zu bekommen. Es ist stattdessen sinnvoller, nur einen Thread zu wecken, welcher dann nach seinem sequentiellen Teil den nächsten weckt. Ein Beispiel für diese Vorgehensweise ist das *Shared/Exclusive-Locking*.

Ein auf Effizienz bedachter Programmierer mag sich hier die Frage stellen, weshalb die Thread-Bibliothek unnötige Kontextwechsel nicht von sich aus unterbindet, hat sie doch bei einem `cond_signal()` oder `cond_broadcast()` Zugriff auf die Information, ob der entsprechende Mutex im Moment frei ist oder nicht. Da jedoch kein Hersteller dieses etwas kompliziertere Protokoll in seiner Dokumentation aufführt, nehmen die Autoren an, daß alle Implementierungen strikt nach Definition arbeiten.

Eine Re-Implementierung der ursprünglich vorgestellten *Shared/Exclusive-Locks* von Bild 2 findet sich in Bild 5. Dabei wurde unter Berücksichtigung der letzten zwei Punkte eine Minimierung der Zahl der Kontextwechsel erreicht. Das Protokoll ist jedoch nach wie vor unfair gegenüber Schreibern.

```
mutex_t mutex;
cond_t awake_reader;
cont_t awake_writer;
int waiting_readers;
int waiting_writers;
int count;    /* Anzahl der Leser oder -1, falls gerade Schreiber aktiv */

void lock_shared(void)
{
  int wr;

  mutex_lock(mutex);
  waiting_readers++;
  while (count < 0)                  /* Wartet nur, wenn gerade */
    cond_wait(awake_reader, mutex); /* ein Schreiber aktiv ist */
  waiting_readers--;
  count++;
  wr = waiting_readers;              /* Muss zwischengespeichert werden */
  mutex_unlock(mutex);
  if (wr > 0)                        /* Ein schonender cond_broadcast() */
    cond_signal(awake_reader);
}

void lock_exclusive(void)
{
  mutex_lock(mutex);
  waiting_writers++;
  while (count != 0)              /* Wartet, bis keiner mehr aktiv ist */
    cond_wait(awake_writer, mutex);
  waiting_writers--;
  count = -1;
  mutex_unlock(mutex);
}

void unlock(void)
{
  int c, ww, wr;

  mutex_lock(mutex);
  if (count < 0)
    count = 0;                    /* unlock() eines Schreibers */
  else
    count--;                      /* unlock() eines Lesers */
  c = count;
  ww = waiting_writers;
  wr = waiting_readers;
  mutex_unlock(mutex);
  if (c == 0)
  {
    if (ww > 0)
      cond_signal(awake_writer);
    else if (wr > 0)
      cond_signal(awake_reader);
  }
}
```

Bild 5 Eine optimierte *Shared/Exclusive-Lock*-Implementierung

4 Vergleich verschiedener Thread-Implementierungen

Für einen einigermaßen umfassenden Überblick über gängige Thread-Systeme auf Arbeitsplatzrechnern wurden folgende Thread-Implementierungen auf die zuvor entwickelten Gesichtspunkte untersucht:

- AIX for PowerPCs v. 4.1
- Java for Linux (development Kit Beta 1.0)
- OS/2 Warp
- PCthreads v. 0.9.2 (User Threads for Linux)
- Solaris Threads v. 2.4
- Windows 95
- Windows NT

Die Untersuchungen basieren auf dem Studium der einschlägigen Dokumentation sowie eigenen Arbeiten und Experimenten mit den aufgeführten Systemen.

AIX-Threads und PCthreads stellen beide eine Implementierung des POSIX-Threads-Standards [IEEE 96] dar, weshalb beide Bibliotheken nahezu identisch sind. Die entscheidende Ausnahme ist ihre Implementierung als Kernel- bzw. User-Threads. Windows 95 und Windows NT stellen beide eine Implementierung der Win32-Schnittstelle dar und sind ebenfalls nahezu identisch.

4.1 Übersicht

4.1.1 Threadmanagement

Tabelle 6 gibt eine erste Übersicht. Oberflächlich betrachtet bieten hier alle Systeme in etwa dieselbe Funktionalität. Bei genauerem Studium entdeckt man jedoch, daß Win32 einem abgebrochenen Thread scheinbar keine Möglichkeit bietet, seine Ressourcen freizugeben. Bei OS/2 gibt es keine Möglichkeit, dem System zu sagen, daß auf einen bestimmten Thread nicht mehr gewartet werden wird (fehlendes `detach()`). Um dennoch eine Speicherlücke zu verhindern, führt OS/2 ein automatisches `detach()` beim Threadende aus. Dies bringt wiederum andere Probleme mit sich: Ein unsynchronisiertes `join()` arbeitet nicht vorhersehbar, ein synchronisiertes ist nicht leicht zu realisieren.

Einige Bibliotheken bieten Erweiterungen gegenüber der vorgestellten üblichen Funktionalität für die Thread-Verwaltung an. Unter *Try-Semantik* verstehen wir eine erweiterte Implementierung, die es dem Programmierer erlaubt, neben dem blockierenden Aufruf auch einen nichtblockierenden zu verwenden, welcher ggf. jedoch mit einer Fehlermeldung zurückkehrt. Unter *Timed-Semantik* verstehen wir eine zusätzliche Erweiterung, welche sich für eine vom Programmierer frei wählbare Zeit (*timeout*) wie die blockierende Variante und danach wie die Try-Variante verhält.

	Java	OS/2	POSIX	Solaris	Win32
`self()`	√	√	√	√	√
`create()`	√	√	√	√	√
`exit()`	√	√	√	√	√
`join()`	tim	try	√	√	tim
`detach()`	gc	–	√	√	√
`cancel()`	√	√	√	√	√
Abbruchmaskierung	–	√	√	√	–
Abbruchvorsorge	√	exc	√	√	exc

√	Implementiert
–	Nicht implementiert
try	Zusätzlich mit Try-Semantik implementiert (siehe Text)
tim	Zusätzlich mit Timed-Semantik implementiert (siehe Text)
gc	Nicht notwendig, Garbage Collection kümmert sich darum
exc	Ausnahme-Cleanup-Handler implementiert, jedoch keine Thread-Ende-Handler

Tabelle 6 Threadmanagement im Überblick

4.1.2 Synchronisation

Bei den Synchronisationsmechanismen zeigen sich die ersten großen Unterschiede zwischen den Systemen (Tabelle 7). POSIX (AIX und PCthreads), Java und Solaris arbeiten mit Monitoren, OS/2 und Win32 hingegen mit Ereignisvariablen.

Die Vielfältigkeit der verschiedenen Implementierungen und eine gewisse Ad-hoc-Herangehensweise der Hersteller offenbaren sich sehr gut bei den Mutexes: Alle drei möglichen Implementierungsvarianten (siehe Abschnitt 2.2) sind vertreten, keine Bibliothek aber erlaubt dem Programmierer die Wahl, welche Version er denn gerne hätte.

Weitere Auffälligkeiten sind:

- Java bietet nur sehr eingeschränkte Monitore.
- OS/2, Solaris und Win32 bieten alle Synchronisationsmittel (außer den kritischen Bereichen) auch prozeßübergreifend an.
- Win32 erlaubt durch sein Objektkonzept eine einheitliche Verwaltung der Synchronisationsmittel. Insbesondere laufen alle blockierenden Aufrufe einheitlich über eine Funktion, der man einen Timeout als Parameter mit übergeben kann.
- OS/2 und Win32 erlauben, auf mehrere Objekte zu warten (entweder auf alle oder auf eines), jedoch ist die OS/2-Funktion sehr stark eingeschränkt.
- Lediglich Solaris bietet über die aufgeführten Synchronisationsmittel hinaus noch das Shared/Exclusive-Locking an, jedoch nur in der für Schreiber unfairen Variante.

	Java	OS/2	POSIX	Solaris	Win32
Mutexes	rec	tim, rec, ip	try, nrec	tim, dl, ip	tim, rec, ip
Bedingungsvariable	1, tim	–	tim	tim, ip	–
Kritische Bereiche	–	mt–, rec	–	–	rec
Semaphore	–	–	–	tim, ip	tim, ip
Ereignisvariable	–	tim, man, ip	–	–	tim, gen, ip
Warten auf mehrere Objekte	–	tim	–	–	tim, mix

–	Nicht implementiert
try	Zusätzlich mit try-Semantik implementiert
tim	Zusätzlich mit timed-Semantik implementiert
rec	Rekursiv implementiert
nrec	Nicht-rekursiv implementiert
dl	Als verklemmende (deadlocking) Mutexes implementiert (Solaris)
ip	Auch als prozeßübergreifender (inter-process) Mechanismus verfügbar
1	Lediglich *eine* Bedingungsvariable pro Mutex möglich (Java)
mt–	Schaltet Multithreading im Prozeß vorübergehend ab (OS/2)
man	Nur manuell rückzusetzende Ereignisvariable implementiert
gen	Allgemeine Ereignisvariable implementiert
mix	Erlaubt gemischte Angabe von Synchronisationsobjekten

Tabelle 7 Synchronisationsmechanismen im Überblick

4.1.3 Weitere Unterschiede

Die übrigen untersuchten Implementierungsunterschiede sind in Tabelle 8 gegenübergestellt. Bis auf die User-Thread-Bibliotheken von Java und PCthreads stellen alle Systeme Kernel-Threads zur Verfügung, Solaris als einziges System sogar beides.

An dieser Stelle sei noch einmal kurz auf den Punkt „Systemstillstand“ eingegangen. Wie in Abschnitt 2 schon erwähnt, hat keines der hier untersuchten Systeme ein Protokoll implementiert, das im Falle von ständig laufenden, statisch hochprioren Threads einen Ausweg aus dem Systemstillstand bietet. Das folgende kleine OS/2-Programm etwa beansprucht alle Rechenleistung für sich, das System reagiert u.a. auf keine weiteren Benutzereingaben mehr.

```
#define INCL_DOS
#include <os2.h>

void main()
{
        DosSetPriority(PRTYS_THREAD, PRTYC_TIMECRITICAL, 31, 0);
        while(1);
}
```

Nachdem Auswegprotokolle also gänzlich fehlen, stellt sich lediglich noch die Frage, *wie* statisch hochpriore Threads erzeugt werden können. Es stellt sich heraus, daß bei Multiuser-Betriebssystemen lediglich Benutzer mit Super-User-Rechten, bei Singleuser-Systemen allerdings jeder dazu in der Lage ist. Es gibt insbesondere keine programmspezifischen Rechte, die der Superuser bzw. der Benutzer vergeben könnte, welche einen eventuellen Systemstillstand verhindern würden.

	AIX	Java	OS/2	PCthreads	Solaris	Win95	WinNT
User/Kernel	K	U	K	bl.U	K+bl.U	K	K
Scheduler	fifo+rr+dyn	impl	rr+dyn/rr	fifo+rr+dyn	spec	rr+dyn	rr+dyn
Systemstillstand	super	any	any	any	super	any	super
Thread-spezifische Daten	√	√	-	√	√	√	√
Rückgabewert?	√	-	-	√	√	√	√
Prozeßende	last	last	init	init	last	init	init
Signalzustellung	mixed	target	mixed	mixed	mixed	?	?
`join()` nach `exit()`	√	√	-	√	√	√	√
Verlorene Mutexes	-	rel	sys	-	-	sys	sys

√	Implementiert
-	Nicht implementiert
K	Kernel-Threads
U	User-Threads
bl.U	User-Threads mit blockierenden Systemaufrufen
dyn	Dynamisches Prioritäts-Scheduling
rr	Statisches Round-Robin-(RR-)Prioritäts-Scheduling
fifo	Statisches FIFO-Prioritäts-Scheduling
impl	Implementationsabhängig: FIFO- oder RR-Strategie (Java)
spec	User: FIFO; Kernel: FIFO+RR+fair (Solaris)
super	Nur Benutzer mit Superuser-Rechten darf Scheduling statisch beeinflussen
any	Jeder Benutzer hat vollen Zugriff auf das Scheduling
last	Ende, wenn der letzte Thread beendet wird
init	Ende, wenn der Initial-Thread beendet wird
rel	Mutexes werden automatisch freigegeben (release)
sys	Das System erkennt, daß der Besitzer beendet ist
target	Signalzustellung an bestimmten Ziel-Thread
mixed	Signalzustellung an andere Threads möglich

Tabelle 8 Weitere Implementierungsunterschiede

4.2 Thread-Implementierungen im Detail

In diesem Abschnitt werden im Detail die Funktionsnamen der jeweiligen Implementierungen wiedergegeben, die in etwa denen von Abschnitt 2 entsprechen. Dies soll einen weiteren Vergleich von Details der verschiedenen Thread-Bibliotheken ermöglichen.

4.2.1 Implementierungen des POSIX-Threads-Standards (AIX 4.1, PCthreads)

Die hier aufgeführten Informationen entstammen dem *AIX 4.1 Programming Guide,* dem *PCthreads Manual Pages* sowie dem POSIX-Threads-Standard [IEEE 96].

Threadmanagement	
`self()`	`pthread_self()`
`create()`	`pthread_create()`
`exit()`	`pthread_exit()`
`join()`	`pthread_join()`
`detach()`	`pthread_detach()` oder mittels Attribut bei `pthread_create()`
`cancel()`	`pthread_cancel()`
Abbruchmaskierung	`pthread_setcancelstate()`, `pthread_setcanceltype()` und `pthread_testcancel()`
Abbruchvorsorge	`pthread_cleanup_push()` und `pthread_cleanup_pop()`

Synchronisation	
Mutexes	Nicht-rekursiv
`mutex_init()`	`pthread_mutex_init()`
`mutex_destroy()`	`pthread_mutex_destroy()`
`mutex_lock()`	`pthread_mutex_lock()` und `pthread_mutex_trylock()`
`mutex_unlock()`	`pthread_mutex_unlock()`
Bedingungsvariable	
`cond_init()`	`pthread_cond_init()`
`cond_destroy()`	`pthread_cond_destroy()`
`cond_wait()`	`pthread_cond_wait()` und `pthread_cond_timedwait()`
`cond_signal()`	`pthread_cond_signal()`
`cond_broadcast()`	`pthread_cond_broadcast()`

`Scheduling`	
`getpriority()`	`pthread_getschedparam()` oder mittels Attribut bei `pthread_create()`
`setpriority()`	`pthread_setschedparam()` oder mittels Attribut bei `pthread_create()`

4.2.2 Threads in Java

Die Informationen sind dem Java-Tutorial des JDK (*Java Development Kit*) entnommen.

Threadmanagement	
`self()`	`currentThread()`
`create()`	Neue Instanz einer Subklasse von *Thread* anlegen oder eine eigene Klasse als *runnable* implementieren, Methode `start()` aufrufen. Der so neu erzeugte Thread führt die Methode `run()` aus.
`exit()`	`stop()`
`join()`	`join()`
`detach()`	Wegen Garbage Collection nicht nötig
`cancel()`	`stop()` (auffangbar) oder `destroy()` (definitiv)
Abbruchmaskierung	Nicht vorhanden
Abbruchvorsorge	Mittels `try {...} finally {...}` oder `try {...} catch(ThreadDeath var) {...}`

`Synchronisation`	
Mutexes	Rekursiv
`mutex_init()`	Nicht vorhanden/nötig
`mutex_destroy()`	Nicht vorhanden/nötig
`mutex_lock()`	Eintritt in `synchronized {...}`
`mutex_unlock()`	Austritt aus `synchronized {...}`
Bedingungsvariable	Nur *eine* pro Mutex
`cond_init()`	`Nicht vorhanden/nötig`
`cond_destroy()`	`Nicht vorhanden/nötig`
`cond_wait()`	`wait()`, Timeout-Angabe möglich
`cond_signal()`	`notify()`
`cond_broadcast()`	`notifyAll()`

`Scheduling`	
`getpriority()`	`getPriority()`
`setpriority()`	`setPriority()`

4.2.3 Threads in OS/2

Die aufgeführten Informationen stammen aus dem *OS/2-Toolkit* von IBM.

Threadmanagement	
`self()`	Mittels `DosGetInfoBlocks()`
`create()`	`DosCreateThread()` bzw. `_beginthread()`, falls der neue Thread C-Bibliotheksfunktionen aufrufen wird
`exit()`	Mittels `DosExit()` bzw. `_endthread()`, falls der Thread mit `_beginthread()` erzeugt wurde
`join()`	`DosWaitThread()`
`detach()`	Keine Funktion vorhanden; automatisches Detach bei Thread-Ende
`cancel()`	`DosKillThread()`
Abbruchmaskierung	`DosEnterMustComplete()` und `DosExitMustComplete()`
Abbruchvorsorge	Nicht vorhanden

Synchronisation	
Mutexes	Rekursiv, Interprozeß-Synchronisation möglich
`mutex_init()`	`DosCreateMutexSem()` bzw. `DosOpenMutexSem()`
`mutex_destroy()`	Letztes `DosCloseMutexSem()` (Prozeßende schließt automatisch)
`mutex_lock()`	`DosRequestMutexSem()`, Timeout-Angabe möglich
`mutex_unlock()`	`DosReleaseMutexSem()`
Kritische Bereiche	Rekursiv, blockiert alle anderen Threads im Prozeß
`crit_init()`	Nicht vorhanden/nötig
`crit_destroy()`	Nicht vorhanden/nötig
`crit_enter()`	`DosEnterCritSec()`
`crit_leave()`	`DosExitCritSec()`
Ereignisvariable	Nur manuelle; Interprozeß-Synchronisation möglich
`event_init()`	`DosCreateEventSem()` bzw. `DosOpenEventSem()`
`event_destroy()`	Letztes `DosCloseEventSem()` (Prozeßende schließt automatisch)
`event_wait()`	`DosWaitEventSem()`, Timeout-Angabe möglich
`event_set()`	`DosPostEventSem()`
`event_reset()`	`DosResetEventSem()`
Warten auf mehrere Objekte	Mittels `Dos*MuxWaitSem()`; nur Objekte *eines* Typs möglich; Timeout-Angabe möglich

`Scheduling`	
`getpriority()`	`Mittels DosGetInfoBlocks()`
`setpriority()`	`DosSetPriority()`

4.2.4 Solaris-Threads

Solaris hat als einziges hier vorgestelltes System sowohl User- als auch Kernel-Threads implementiert. Realisiert wurde dieses Konzept durch ein dreischichtiges Modell: Der Anwendungsprozeß teilt seine Arbeit auf *User-Threads* auf (in Solaris-Terminologie: Threads), eine Zwischenschicht bildet diese auf *Kernel-Threads* ab (in Solaris-Terminologie: Light Weight Processes, LWPs), welche dann vom Kernel auf die vorhandenen Prozessoren aufgeteilt werden. Die Nachteile der traditionellen User-Threads fallen bei diesem System weg, da hier die beiden Scheduler (zum einen der von User-Threads auf einen Kernel-Thread, zum anderen der von Kernel-Threads auf einen Prozessor) miteinander kommunizieren können. Graphisch ist dieses Konzept in Bild 6 dargestellt.

Der Programmierer kann durch Zuhilfenahme der Funktionen `thr_getconcurrency()` und `thr_setconcurrency()` die Zwischenschicht steuern, indem er die Anzahl der LWPs für die *freien* Threads in seinem Prozeß setzt. Neben den freien (*unbound*) Threads gibt es auch solche, die einem bestimmten LWP eindeutig zugeordnet sind (*bound threads*).

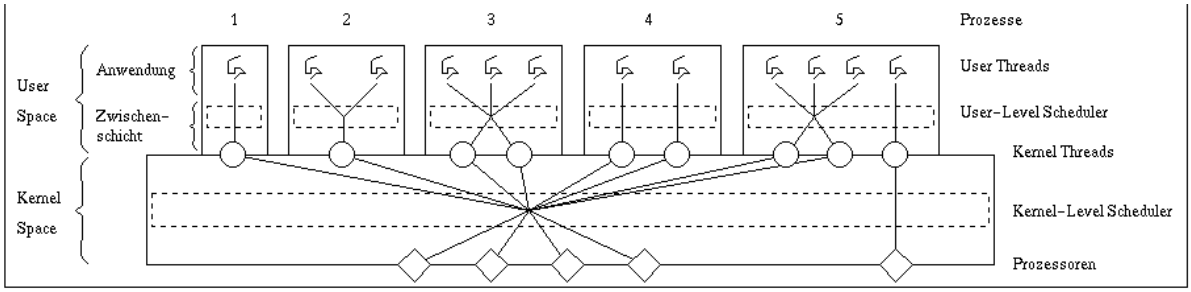

Bild 6 Solaris Threads

Die Informationen zu Solaris-Threads entstammen dem *Solaris Programming Guide*.

Threadmanagement	
`self()`	`thr_self()`
`create()`	`thr_create()`
`exit()`	`thr_exit()`
`join()`	`thr_join()`
`detach()`	Mittels Flag bei `thr_create()`
`cancel()`	`thr_kill()`
Abbruchmaskierung	Mittels `thr_sigsetmask()`
Abbruchvorsorge	Mittels Signal-Handler

Synchronisation	
Mutexes	Verklemmend, Interprozeß-Synchronisation möglich
`mutex_init()`	`mutex_init()`
`mutex_destroy()`	`mutex_destroy()`
`mutex_lock()`	`mutex_lock()`, Timeout-Angabe möglich
`mutex_unlock()`	`mutex_unlock()`
Bedingungsvariable	Interprozeß-Synchronisation möglich
`cond_init()`	`cond_init()`
`cond_destroy()`	`cond_destroy()`
`cond_wait()`	`cond_wait()`, Timeout-Angabe möglich
`cond_signal()`	`cond_signal()`
`cond_broadcast()`	`cond_broadcast()`
Semaphore	Interprozeß-Synchronisation möglich
`sema_init()`	`sema_init()`
`sema_destroy()`	`sema_destroy()`
`sema_p()`	`sema_wait()`, Timeout-Angabe möglich
`sema_v()`	`sema_post()`
Leser/Schreiber-Lock	Unfair gegenüber Schreiber
`rw_lock_init()`	`rw_lock_init()`
`rw_lock_destroy()`	`rw_lock_destroy()`
`rw_rdlock()`	`rw_rdlock()`, Timeout-Angabe möglich
`rw_wrlock()`	`rw_wrlock()`, Timeout-Angabe möglich
`rw_unlock()`	`rw_unlock()`

`Scheduling`	
`getpriority()`	`thr_getprio()`
`setpriority()`	`thr_setprio()`

4.2.5 Win32-Threads (Windows 95, Windows NT)

Die folgenden Informationen sind [Richter 97] entnommen.

Threadmanagement	
`self()`	Mittels `GetCurrentThread()` *und* `DuplicateHandle()`
`create()`	`CreateThread()`
`exit()`	`ExitThread()`
`join()`	`WaitForSingleObject()`, Ergebnis mit `GetExitCodeThread()`
`detach()`	Letztes `CloseHandle()` (Prozeßende schließt automatisch)
`cancel()`	`TerminateThread()`
Abbruchmaskierung	Nicht vorhanden
Abbruchvorsorge	Nicht vorhanden

Synchronisation	
Mutexes	Rekursiv, Interprozeß-Synchronisation möglich
`mutex_init()`	`CreateMutex()`
`mutex_destroy()`	Letztes `CloseHandle()` (Prozeßende schließt automatisch)
`mutex_lock()`	`WaitForSingleObject()`, Timeout-Angabe möglich
`mutex_unlock()`	`ReleaseMutex()`
Kritische Bereiche	Rekursiv
`crit_init()`	`InitializeCriticalSection()`
`crit_destroy()`	`DeleteCriticalSection()`
`crit_enter()`	`EnterCriticalSection()`
`crit_leave()`	`LeaveCriticalSection()`
Semaphore	Interprozeß-Synchronisation möglich
`sema_init()`	`CreateSemaphore()` bzw. `OpenSemaphore()`
`sema_destroy()`	Letztes `CloseHandle()` (Prozeßende schließt automatisch)
`sema_p()`	`WaitForSingleObject()`, Timeout-Angabe möglich
`sema_v()`	`ReleaseSemaphore()`
Ereignisvariable	Allgemeine (d.h. manuelle und automatische) verfügbar; Interprozeß-Synchronisation möglich
`event_init()`	`CreateEvent()` bzw. `OpenEvent()`
`event_destroy()`	Letztes `CloseHandle()` (Prozeßende schließt automatisch)
`event_wait()`	`WaitForSingleObject()`, Timeout-Angabe möglich
`event_set()`	`SetEvent()`
`event_reset()`	`ResetEvent()`
Warten auf mehrere Objekte	`WaitForMultipleObjects()` statt `WaitForSingleObject()`; gemischte Objekttypen und Timeout-Angabe möglich

`Scheduling`	
`getpriority()`	`GetPriorityClass()` und `GetThreadPriority()`
`setpriority()`	`SetPriorityClass()` und `SetThreadPriority()`

5 Zusammenfassung

In diesem Artikel wurden die wesentlichen Konzepte von Thread-Bibliotheken wie Thread-Management, Synchronisationsmittel und Scheduling, einige Implementierungsaspekte sowie Hinweise zur Programmierung mit Threads zusammengefaßt. Anhand der wichtigsten Gesichtspunkte und Funktionalität wurden anschließend einige gängige Thread-Systeme, die heute auf Arbeitsplatzrechnern verfügbar sind, einander gegenübergestellt. Dabei zeigten sich im Detail doch erhebliche Unterschiede zwischen den verschiedenen Systemen, die neuen Einarbeitungsaufwand beim Wechsel zu einem anderen Betriebs- oder Thread-System und Sorgfalt bei der Programmierung erfordern. Besonders auffällig sind die Unterschiede bei den zur Verfügung gestellten Synchronisationsmechanismen und deren Implementierungen.

Dies zeigt, daß sich die Konzepte von und die Programmierung mit Threads noch in Entwicklung befinden. Auch mit dem POSIX-Threads-Standard [IEEE 96] dürfte hier noch kein Ende erreicht sein. Nicht berücksichtigt in dem Vergleich sind mögliche Leistungsunterschiede der betrachteten Thread-Bibliotheken. Es ist zu erwarten, daß sich auch diesbezüglich signifikante Unterschiede ergeben.

Literaturverzeichnis

[Birrell 91] A.D. Birrell, „An Introduction to Programming with Threads“, in: *Systems Programming with Modula-3*, Greg Nelson (ed.), Prentice Hall 1991.

[Hoare 74] C.A.R. Hoare, „Monitors: An Operating System Structuring Concept“, *Comm. ACM*, 17(10), 549-557, Oct. 1974.

[IEEE 96] IEEE Computer Society, *Information technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]*, International Standard ISO/IEC 9945-1 - ANSI/IEEE Std 1003.1, Second Edition, IEEE 1996.

[Kleiman et al. 96] S. Kleiman, D. Shah, B. Smaalders, *Programming with Threads*, SunSoft Press / Prentice Hall 1996.

[Richter 97] J.M. Richter, *Microsoft Windows - Programmierung für Experten,* ISBN 3-86063-336-8, 1997.

[Tanenbaum 92] A.S. Tanenbaum, *Modern Operating Systems*, Prentice Hall 1992.