1 NUTZUNG VON THREADS

LynxOS (ab Version 2.0) unterstützt ein Prozeß-Modell, das eine Erweiterung zum traditionellen UNIX-Prozeßmodell ist. Ein traditioneller UNIX-Prozeß ist begrenzt auf eine Aktivität, die zur Zeit abläuft. Man bezeichnet dieses Modell als „Singlethreaded Prozeß-Modell. Im Gegensatz dazu unterstützt das Lynx Prozeß-Modell mehrere Aktivitäten, die innerhalb eines einzelnen Prozesses ablaufen können: Ein Prozeß unter LynxOS kann multithreaded sein. Die Fähigkeit mehrere Threads abarbeiten zu können ist eine leistungsfähige Eigenschaft bei der Behandlung der natürlichen Parallelität, wie sie in Echtzeit-Systemen vorkommt. Die Programmierung in mehreren Threads ist für den Programmierer einfacher zu handhaben, als wenn er versucht die parallelen Aktionen mit einem „singlethreaded“ Prozeßmodell zu behandeln.
Threads erfordern ein Umdenken bei der Programmerstellung. Diese Text verfolgt das Ziel, zu erläutern warum und wie Threads in Real-Time-Applikationen zu verwenden sind.
Die Betrachtung von Threads erfordert einen größeren Platz, deshalb ist diese Dokumentation entsprechend umfangreich.
Folgende Punkte werden beschrieben:

- Anforderungen und Modelle für Thread-Anwendungen.
- POSIX Threads im Detail
- Wie beeinflußt das Vorhandensein von Threads die existierenden UNIX-Eigenschaften?
- Vorschläge für das Programmieren mittels Threads
- Vergleich der Lynx's Threads mit den POSIX-Threads

Lynx Threads bieten eine erweiterte Funktionalität gegenüber den POSIX Threads. Lynx Threads und POSIX Threads kann man im Prinzip gemischt aufrufen, obgleich die Benutzung von Lynx-Aufrufen isoliert sein sein, wenn es möglich ist, um eine POSIX-Kompatibilität ihrer Applikation zu erhalten.
Die POSIX-Eigenschaften sind im Detail in den „manual pages“ zu den einzelnen Funktionen oder in [P4aD4,P4D9]erklärt.

1.1 Threads für Echtzeit Programmierung

1.1.1 Was sind Threads? Warum werden sie gebraucht?

Echtzeit-Systeme übernehmen, im allgemeinen, gleichzeitig die Überwachung und Steuerung von mehren Aktionen. Mit anderen Worten, viele Dinge laufen parallel ab. Mögliche Alternativen, um sich mit dieser Parallelität unter UNIX auseinanderzusetzen, sind die Verwendung mehrerer Prozesse, dem ad-hoc Multiplexing eines einzelnen Prozesses in mehrere, ziemlich unabhängige Tasks und die Verwendung von UNIX-Signalen und asynchronen Systemeigenschaften.
Keine von diesen Methoden ist zufriedenstellend. Das Grundproblem ist das Einbringen der natürlichen Parallelität der Echtzeit-Verarbeitung in das nichtparallele und damit eigentlich ungeeigneten UNIX Modell. (Das ist so, als wenn man einen runden Pfock in eine viereckige Öffnung zwängt).
Es ist wichtig in einer korrekten und robusten Applikation, die Dinge einfach zu halten. Wenn dann der Programmierer aber mit der Programmierung des zugrundeliegenden Betriebssystems zu kämpfen hat, stellt das eine weitere Erschwerung der Programmierung dar. Ein Prozeß-Modell, das die naturgemäße Parallelität von Echtzeit nutzt, macht die Arbeit von Echt-Zeit-Programmierern einfacher und erleichtert die Wartung, das Verständnis und das Debuggen.

1.1.1.1 Threads und die Parallelität in Echtzeitsystemen?

Einen typischen UNIX-Prozeß kann man sich als einen Adreßraum vorstellen, der geschützt von allen anderen Adreßräumen des Systems ist, in dem ein „Ding“ ausgeführt wird. Dieses „Ding“ ist bekannt als Ausführungs-Thread oder einfacher als Thread. Was ist das „Ding“ ? Es ist eigentlich ein Maschinenzustand: ein Satz von Registern.
Wie würde man mit diesem Modell vorgehen, um drei separate, konkurrierende Tasks ablaufen zu lassen? Es gibt es verschiedene Möglichkeiten.
Ein Weg ist die Erzeugung von drei separaten UNIX-Prozessen. Das Problem hierbei ist der geschützte Adreßraum. Die drei Prozesse sind voneinander getrennt. Die Kommunikation wird schwierig und ist im begrenzten Rahmen möglich.
Ein anderer Weg wäre, den Code von allen drei Task in einen einzelnen Prozeß zu legen und innerhalb von diesem Prozeß von einen Job zum anderen zu springen in Abhängigkeit von der Zeit und der Arbeit die gerade auszuführen ist. Bei genauerer Betrachtung, ist diese Vorgehensweise ein Eingeständnis, daß das Prozeß-Modell, das vom Betriebssystem bereitgestellt wird, unangemessen ist.
Was ist passiert? Du legst drei Jobs in einen Prozeß. Das heißt, du mußt genau festlegen was jeder Thread (Job) machen darf. Sie dürfen, um Kollisionen zu vermeiden, ihren timeslot (Zeitbereich) nicht verlassen.

1.1.1.2 Das richtige Prozeß-Modell

Die Eigenschaften der Lynx Threads bieten das Erforderliche: mehrere Threads innerhalb eines einzelnen Adreßraumes. Die Threads sind separate Kerneinheiten: Jeder kann Systemcall's aufrufen und blockieren, ohne die anderen Threads zu blockieren. Falls ein Thread mit einer höheren Priorität als der laufenden Thread bereit wird, dann verdrängt der Thread den gerade laufenden, ebenso wie bei Prozessen unter LynxOS**.**

1.1.1.3 Zwei Schnittstellen zu Threads

LynxOS 2.0 unterstützt zwei unterschiedliche Schnittstellen zu Threads. Zum einen ist das die lynxeigene proprietäre Schnittstelle, genannt STS oder System-Threads, die eine low-level Schnittstelle zu Threads darstellt. Und zum anderen werden Threads entsprechend den Festlegungen in POSIX 1003.4a, Draft 4 unterstützt.
Die manipulierten Threads haben die gleichen zugrundeliegende Systemeigenschaften. Man kann sogar diese beiden Interfaces in einem einzelnen Programm vermischen. Es ist aber nicht ratsam, dieses ohne zwingende Gründe zu tun, da dieses nachteilig für die Anwenderportabilität und Verständlichkeit wäre. Wenn aber diese zwingenden Gründe vorliegen, wird die Verwendung der POSIX 1003.4a-Thread-Eigenschaften empfohlen, da somit eine größere Portabilität erreichbar ist.

1.2 Die Grundlagen von Threads

Ein typischer UNIX Prozeß besteht aus einem einzelnen Thread, der in einem geschützten virtuellen Adreßraum liegt. Der Prozeß hat verschieden Merkmale, die mit ihm verbunden sind: Files, single actions, ein Nutzer ID und ein Gruppen-ID, IPC-Descriptoren und der gleichen.
Threads erweitern dieses Modell in eine Richtung. Während ein typischer UNIX Prozeß nur eine Tätigkeit zu einer Zeit ausführt, kann ein multithreaded-Prozeß viele Tätigkeiten parallel abarbeiten. Einem Thread wird nach den selben Regeln der Prozessor zugeteilt, wie einem Prozeß. Jedoch werden diese Threads alle im gleichen Adreßraums ausgeführt, sie verwenden dieselben Files, haben die gleiche Nutzer ID, Gruppen ID und so weiter.

1.2.1 Was ist mit den Prozessen ?

Im Multithread-Modell sind alle Threads des Systems in einem großen Pool mit allen anderen Prozessen zusammen(single-threaded „Prozesse“ wie `ls` und `cc` und `rlogind`) und werden

relativ zueinander scheduled. Prozeß-Prioritäten und Thread-Prioritäten spielen zusammen: Ein Thread mit der Priorität 20 wird einen single-threaded Prozeß mit der Priorität 15 unterbrechen, und umgedreht wird einen Thread mit der Priorität 31 einen Thread mit der Priorität 20 unterbrechen.
Eine gewisse Konfusion kann häufig hinsichtlich der Unterscheidung zwischen Threads und Prozessen entstehen. Nämlich wo befindet sich der Prozeß, wenn ein Programm in einer multithreaded Art geschrieben wurde? Was macht der Prozeß, bei drei existierenden Threads, die alle etwas anderes machen?.
Die Antwort ist: Der Prozeß macht überhaupt nichts. Der Prozeß ist nichts weiter als ein passiver Behälter für Threads. Ich finde es nützlich einmal in folgender Art über das System nachzudenken: Es gibt keine Prozesse mehr, sondern nur noch Threads. Was früher ein Prozeß war, sprich ein Aufruf von `ls`, ist jetzt nichts anderes als ein einzelner Thread, der in einem Adreßraum läuft. Für den Prozeß bleibt jetzt nur noch ein konventioneller „Handle“ für einen Adreßraum, ein Satz von File-Deskriptoren und dergleichen. Der Prozeß an sich ist einfach ein Container für Threads, der selbst nicht läuft. Die Threads im Innern des Prozesses sind das, was scheduled wird.

1.2.1.1 Einige Überlegungen

Als die Threads in LynxOS einfügt wurden, wurde ein Weg benötigt, um alle Threads beobachten zu können. Deshalb wurde das `ps`-Utility erweitert. Wenn man `ps xat` eingibt, erhält man neben der normalen `ps`-Ausgabe eine Spalte mit den Thread-ID's von sogenannten laufenden Prozessen im System. Die „t-“Option bezieht sich auf die Thread-ID's. Beachte, daß es für jeden Prozeß mindestens einen Thread mit der ID-Nummer 32767 gibt. In der aktuellen Version hat ein Thread in jedem Prozeß diese ID (Das kann in nachfolgenden Versionen ohne Notiz geändert werden. Ihre Applikationen sollten sich nicht auf diese Nummer beziehen.) Hinzukommende Threads in einen Prozeß haben andere ID-Nummern

1.2.1.2 Threads und Stream Tasks

Frühere Versionen von LynxOS führten das Konzept von "Stream Tasks" für priorisierte Interruptzugriffe ein. Diese Stream Tasks werden innerhalb des Kerns als Antwort auf einen bestimmten I/O Stream (daher der Name) ausgeführt. Diese Stream-Task behandelt Interruptzugriffe, die in einem normalen UNIX System die Interrupt-Service-Task ausführen würde. Das erlaubt LynxOS den Interruptzugriff zu priorisieren und ist ein Schlüssel zu für die excellente **„dispatch latency“.**
Threads und Stream Tasks werden von der gleichen Systemeinheit erzeugt . Wenn man „`ps xatT`“ eingibt, sieht man nicht nur den Thread ID vom laufenden Prozeß des Systems, sondern, daß es auch eine extra Spalte für jene Stream-Tasks gibt, die gerade auf dem System laufen.

1.2.2 Wie laufen Threads ?

Wenn ein neuer Prozeß das erste Mal gestartet wird (als Resultat von fork()- und exec()-Calls), hat er nur einen Thread in sich, der als Initialisierungsthread bezeichnet wird. Dieser Initialisierungsthread beginnt gewöhnlich in einem Programm bei main(). Der Initialisierungsthread kann einen Call ausgeben, um einen neuen Thread zu erzeugen. Ab diesem Punkt werden zwei Threads in diesen Prozeß laufen. Das erste Thread sagt dem System, an welcher Stelle der neue Thread beginnen soll und wieviel Stack er benutzen wird. Auf diese Weise können neue Threads erzeugt werden und an verschiedenen Stellen im System laufen.
Da Threads andere Threads erzeugen, so wie UNIX-Prozesse mit fork und exec andere Prozesse erzeugen, kann man versucht sein zu denken, daß eine Hierachie von Threads existiert. Schließlich, hat im Umfeld der UNIX-Prozesse jeder Prozeß einen Vaterprozeß.

Geschwisterprozesse werden vom gleichen Elternprozeß kreiert. Diese Hierachie stellt verschiedene Eigenschaften bereit, die nützlich für die Gruppierung von Prozessen ist.
Das ist nicht der Fall bei Threads. Bei Lynx gibt es kein semantisches Gewicht, das zurückzuführen ist auf einen einen Thread kreierenden Thread. Alle Threads in einen Prozeß sind völlig gleichberechtigt. Es ist unwichtig welcher Thread welchen erzeugt hat.

1.2.3 Blockieren

Ein Thread einer Applikation rechnet die Quadratwurzel von Zwei mit größtmöglicher Genauigkeit aus. Dazu ist eine Endlos-Schleife notwendig. Ein zweiter Thread möchte einige Daten von der Festplatte holen. Was passiert, wenn der zweite Thread den Kernel blockiert, während er auf die Daten wartet? Das Erhoffte passiert. Das lesende Thread ist blockiert und der erste Prozeß arbeitet unvermindert weiter. Dieses Beispiel steht im Widerspruch zu der Situation, bei der jemand versucht, mehrere Aktivitäten in einem einzelnen Thread zu multiplexen. Das steht auch im Kontrast zu einigen Threadpaketen, die verfügbar bei einigen Anbietern sind (z.B. Sun´s LWP-Library). Diese Threadpakete werden auch, weil sie ganz außerhalb des Kernels implementiert sind „Library Threads“ oder „User-Level Threads“ genannt, da sie außerhalb des Kernels im Bibliotheks-Code implementiert sind.
Da der Kernel im System nichts über separate Threads weiß, wird ein Blockieren eines Threads alle Treads in einer Anwendung blockieren.

1.2.4 Stack und Speicher

Alle Threads eines Prozesses teilen sich einen Speicherbereich. Dieses hat zur Folge, daß ein Thread auf die Daten von jedem anderen Thread innerhalb eines Prozesses zugreifen könnte. Es ist also beim Programmieren von Threads darauf zu achten, daß diese nicht auf die selbe Art und Weise auf einzelne Datenstruckturen zugreifen. Beim Stack eines Threads liegt die Sache allerdings etwas anders.
Obwohl die Stack's im gleichen Adressraum liegen, liegen sie doch auf verschiedenen Adressen. Die Stackgröße muß bereits beim Kreieren des Threads angegeben werden. Der Thread ist dann in der Lage seinen Thread entsprechend dieser Größe auszudenen, aber nicht weiter. Die Threadstack's befinden sich am oberen Ende des Adressraums einer Applikation und werden durch einige „unmapped pages“ voneinander getrennt. Dieses soll verhindern, daß der Stack eines Threads in den Stack eines anderen Threads überläuft. Es ist aber möglich diese Schutzvorrichtung zu umgehen.
Beim 80386, beim PS/2 und beim MVME-147 umfassen z.B. zwei Pages diese ungemappten Bereiche. Das sind 8192 Bytes des Typs „Faß mich nicht an“´. Falls der laufenden Thread in diesen Bereich beim Zugriff auf eine große Stackvariable (z.B. ein Array von 10000 Character-Werten) hineinläuft und dann versucht auf weitere Stackvariablen zuzugreifen, dann kann es passieren, daß er den Stack des nächsten Threads überschreibt. Also man muß höllisch aufpassen.
Obwohl die Threadstacks sich auf verschiedenen Adressen befinden, liegen sie im selben Adreßraum. Das ist wichtig. Ein Thread kann auf den Stack eines anderen Threads kontrolliert oder unkontrolliert zugreifen. In einem einfachen Beispiel sind zwei Threads (t1, t2) kreiert, die jeweils die Funktionen f1() und f2() ausführen. Auf einen globalen Pointer können beide Threads zugreifen. Angenommen t1 und t2 haben dieselbe Priorität und t1 läuft vor t2.

```
char *global_pointer = NULL

/* Function executed by one thread, t1 */
f1()
{
            char t1_buf[4096];

            sprintf (t1_buf,“This is T1´s buffer /n“);
```

```
            global_pointer = &t1_buf;
            yield();
}

/* Function executed by another thread, t2 */
f2()
{
            char t2_buf[4096];

            sprintf (t2_buf,“This is T2´s buffer /n“);
            if (global_pointer)
                  printf(global_pointer);
            else
                  printf(“failure: global_pointer not set \n“);
}
```

Die Ausgabe dieses Beispiels wird sein:

```
This is T1´s buffer
```

da der globale Pointer auf den Puffer auf T1´s Stack zeigt.

1.2.4.1 Stack Overflow Handling

Wenn der Stack eines Threads in die unmapped pages überläuft, erhält er ein SIGSEGV-Signal. Das Aussenden und die Behandlung eines Signals benötigt auch Stack. Deshalb alloziert Lynx einige ungemappte Seiten. Falls der Thread aus seinem zugewiesenen Stack rausläuft, dann wird eine Extra-Page im ungemappten Bereich alloziert, in dem die Signalbehandlung von SIGSEGV erfolgen kann. Es ist nur erlaubt, diesen Extrabereich für das Signalhandling zu verwenden.
Wenn der Stack des Threads überläuft, während das Signal-Handling abläuft, dann wird der Prozeß mit dem Signal SIGKILL beendet.

1.3 Programmieren von auf Threads basierenden Programmen

Um ein Programm zu compilieren, das Threads verwendet, müssen beim Aufruf des Compilers die Optionen „-X“ und „-m“ angegeben werden. Die „-X“Option bewirkt, daß die POSIX-Bibliotheken eingebunden werden; durch die Angabe von „-m“ wird die multi-threaded Version der Standardfunktionen (wie z.B. *malloc* und *printf*) verwendet, die intern sich anders verhält als die single-threaded Version.
Falls ein Programm, das nur die proprietären Funktionen von Lynx verwendet, compiliert werden soll, dann kann man ohne die Option „-X“ auskommen, aber es könnte später sehr schwierig werden, die POSIX-Bibliotheken einzubinden, wenn das erforderlich ist.

1.3.1 Lynx Threads und POSIX Threads

Die Version der POSIX-Threads, die in Lynx implementiert sind, entsprechen dem vorläufigen Draft 4 des POSIX Standards 1003.4a. Möglicherweise wird ein späterer Draft zu einem Standard erklärt. [1] Lynx- Corp. unterstützt voll diesen Draft und wird bei einer

[1] Das Thread-Interface POSIX 1003.4a wurde 1994 als neuer POSIX-Standard 1003.1c verabschiedet. Damit wurde der 14. Draft als Standard beschlossen. (Anm. F.G.)

Verabschiedung des offiziellen Standards die Anwender beim Übergang vom frühen Draft zum Standard unterstützen.[2]
Obwohl die Basisfunktionalität der POSIX-Threads ausreichend ist, weist sie doch einige Lücken auf. Diese Lücken werden mit der zugrundeliegenden lynxeigenen proprietären Thread-Implementierung adressiert. Die POSIX- Draft 4-Threads sind in LynxOS als einfache Library Calls zu den lynxeigenen zugrundeliegenden Threads implementiert. (diese werden als System Threads oder einfach sts bezeichnet). Folglich ist es dasselbe einen Thread mittels *pthread_create*- oder mittels der proprietären *st_build*- Funktion zu erzeugen, d.h. beide Threads können mittels der POSIX-Thread-Rufe oder Lynx-Thread-Rufe bearbeitet werden. Folglich sind „POSIX-Threads“ und „Lynx-Threads“ gleich. Es gibt zwei separate Interfaces, die sich mit Threads beschäftigen. Das Lynx-Thread-Interface ist ein bißchen allgemeiner und ein bißchen einfacher.
Aus praktischer Sicht empfehlen wir, möglichst die POSIX-Eigenschaften zu verwenden und jede Verwendung des Lynx-Interfaces hervorzuheben und gut zu dokumentieren. Das wird die Portabilität Ihres Codes erhöhen.

1.4 Verwendung der POSIX-Threads-Funktionen

Dieser Abschnitt gibt einen Überblick über die POSIX-Thread-Funktionen. Die proprietären Thread- Funktionen von Lynx, werden dort, wo sie den POSIX-Standard erweitern, in einem späteren Abschnitt betrachtet. Wenn weitere Informationen notwendig sind, sollte in den Manual-Pages oder im POSIX-Standard selbst nachgesehen werden. [2]

1.4.1 Allgemeine Merkmale der POSIX-Funktionen

- Headers: Um eine der POSIX-Thread-Funktionen zu verwenden, muß <pthread.h> in das C-Programm eingebunden werden.
- Attributes Structures: Im allgemeinen werden beim Aufruf der POSIX-Funktionen bestimmte Dinge erzeugt, wobei Attributstrukturen (attribute structure) verwendet werden, in denen verschiedene Felder gesetzt werden können. Diese Attributen spezifizieren, welche Objekteigenschaften gewünscht sind. Z.B. ist es erforderlich, daß beim Kreieren eines Threads vorher eine Attributstruktur kreiert und initialisiert wird. Eines der Felder in dieser Struktur gibt die gewünschten Stackgröße für den zu kreierenden Thread an; ein anderes zeigt das gewünschte Schedulingattribut des neu erzeugten Threads an. Es gibt Attribut-Strukturen, die beim Kreieren von Threads, Mutexen, Condition Variables verwendet werden.

1.4.2 Kreieren eines Threads: *pthread_create()*

Ein Thread kreiert einen anderen mittels:
`pthread_create(tidp, attr, start_routine , arg);`
Der ID des neu erzeugten Threads steht nach erfolgreicher Ausführung des Systemrufes in `tidp`. Pthread_create() benötigt eine Attributstruktur, welche einiger der folgenden Felder definiert:

- pthread_attr_stacksize
- pthread_attr_inheritsched
- pthread_attr_prio
- pthread_attr_sched

Die Attributstrukur wird mittels *pthread_attr_create()* kreiert und wird mit Default-Werten initialisiert: Die Stackgröße beträgt 8192 Bytes, `inheritsched` wird auf „wahr“ gesetzt, was bedeutet, daß ein Thread, der mit dieser Strukur kreiert wird, die Priorität und das Schedulingverfahren des erzeugenden Threads erbt.
Ein Thread, der mittels *pthread_create()* erzeugt wurde, ist gleich **runnable** und wird, basierend auf seinen Schedulingattributen, zu dem Zeitpunkt in das Scheduling eingereiht,

[2] Die gleichen Aussagen treffen für den Basis-Real-Time-Standard POSIX 1003.4 zu, dessen Draft 9 in LynxOS implementiert ist.

wenn er kreiert ist. Dann wird die Routine, die bei `start_routine` beginnt, mit dem Argument `arg` ausgeführt. Wenn ein Thread mit einer höheren Priorität als die des erzeugenden Threads kreiert wird, dann wird der neue Thread gestartet bevor er vom Systemruf *pthread_create()* zurückkehrt. (Zumindest gilt das bei einem Universalprozessor. Bei einem Multiprozessor sind die Dinge ein wenig komplizierter).
Pthread_attr_create(attr) wird verwendet, um die Thread-Attribut-Struktur mit Default-Werten zu initialiseren. Das ist der durch POSIX anerkannte Weg eine Attributstruktur zu initialisieren. Unter LynxOS 2.0, setzt diese Funktion `pthread_attr_stacksize` auf THREAD_DEFAULT_STACK und setzt die Schedulingattribute auf „Scheduling vom kreierenden Thread erben“ (`pthread_attr_inheritsched`=1). Um neue Werte in der Thread-Attribut-Struktur zu setzen, sollen *pthread_attr_setstacksize(attr,stacksize) pthread_attr_setprio(attr,prio), pthread_attr_setsched(attr,sched), pthread_attr_setinheritsched(attr,inherit) verwendet werden* .

1.4.2.1 Thread Schedulingbereich

Es gibt ein zusätzliches Element innerhalb der Thread-Attributstruktur, das durch POSIX definiert ist: `pthread_attr_scope`. Dieses Element definiert den Bereich (scope) innerhalb dessen der Thread scheduled wird. POSIX-Threads haben entweder einen globalen oder einen lokalen Bereich. Lokaler Bereich: Threads werden relativ zu anderen Threads innerhalb eines Prozesses scheduled, aber es können bezüglich des Schedulings relativ zu anderen Prozesse (und den darin enthaltenen Threads), die gerade laufen, keine Garantien gegeben werden. Globaler Bereich: Threads werden im Wettkampf mit allen anderen Threads innerhalb des Systems scheduled.
In LynxOS 2.0 werden alle Threads global scheduled (globally scoped threads), da nach Ansicht der Lynx-Entwickler nur wenig Sinn für einen lokalen Bereich in einem Real-Time-System gesehen wird.
Das Element `pthread_attr_scope` wird vom System ignoriert, da somit Threads relativ zu allen anderen Prozessen korrekt scheduled werden.

1.4.3 Beendigung eines Threads: *pthread_exit()*

Wenn ein Thread innerhalb eines Prozesses seine Aufgabe beendet hat und sich beenden soll, dann wird *pthread_exit(val)* verwendet. Pthread_exit ist das Threadäquivalent zu exit(). Der Wert der an *pthread_exit* übergeben wird, ist der Exit-Wert des Threads und kann an andere Threads mittels *pthread_join()* übergeben werden.

1.4.3.1 Vergleich von Thread-Exit und Process-Exit

Es gibt einen grundlegenden Unterschied zwischen *pthread_exit()* und *exit(). Pthread_exit* bewirkt, daß nur der aufrufende Thread beendet wird. Wenn ein Thread *exit()* aufruft, werden alle anderen Threads innerhalb des Prozesses beendet. Exit() ist „the end of the world“, insofern ein Prozeß beteiligt ist. *Pthread_exit()* beendet einfach nur den aufrufenden Thread.
Wenn ein normaler single-threaded Prozeß in UNIX seine Main-Routine beendet, dann ist das so, als wenn dieser Prozeß am Ende exit() mit dem Rückgabewert (return value) von `main` aufgerufen hätte. Ebenso gilt, wenn ein Thread von seinem Aufruf zurückkehrt (derjenige Thread dessen Adresse in *pthread_create()* angegeben wurde), dann ist es so, als wenn *pthread_exit()* aufgerufen wurde mit dem Rückgabewert (return value) der Routine als Argument für *pthread_exit().*
Eine Ausnahme von dieser Regel ist der Initial-Thread des Multi-Threads-Progamms. Falls er rückkehrt von seinem Initial-Aufruf aus main(), dann wird der Prozeß beendet, wie im single-threaded Fall.

(*Pthread_exit(val)* überträgt seinen Rückgabewert an den Thread/Prozeß, von dem aus er kreiert wurde. Wird Pthread_exit am Ende eines Inital-Threads aufgerufen, dann verhält sich *Pthread_exit()* wie *exit()* und beendet den Prozeß)

nur Initial-Thread	Initial-Thread kreiert weiteren Thread	nur Prozeß (single-threaded)	Rückkehr ohne Aufruf einer Exit-Funktion (*exit()* bzw *pthread_exit()*) Sonderfall zu Spalte 2

main()
{
...
pthread_exit();
oder
exit()
oder
gar nichts
}
	main()
	{
	...
	pthread_create(t1,...);
	...
	exit();
	}
	
	t1()
	{
	...
	pthread_exit(beliebigerWert)
	oder
	gar nichts
	}
		main()
		{
		...
		exit()
		oder
		gar nichts
		}
			main()
			{
			...
			pthread_create(t1,...);
			}
			
			t1()
			{
			...
			}

Eine letzte Regel: Wenn alle Threads in einem Thread mittels *pthread_exit()* beendet werden oder (Zurückkehren von ihrem Aufruf), dann endet der Prozeß, wenn die letzte Routine, in der ein Thread beendet wird, anstelle von *pthread_exit() exit()* aufrufen würde. D.h. wenn es keine weiteren Threads innerhalb eines Prozesses gibt, dann endet der Prozeß mit dem Exit-Wert des letzten sich beendenden Thread.

1.4.4 Warten auf das Ende eines Threads: *pthread_join()*

Ein Thread kann auf einen anderen warten, um sich dann zu beenden. Die äquivalenten Rufe für Prozesse sind `wait()` und `waitpid()`. *Pthread_join(tid,valp)* wird verwendet, um auf die Beendigung seiner Ausführung eines spezifischen Threads zu warten. Falls der spezifizierte Thread schon beendet ist, dann kommt der Call unmittelbar zurück. Auf einen beendeten Thread kann willkürlich oft gewartet werden; *Wait()* funktioniert innerhalb eines Prozesses nur einmal. Der Exit-Wert des beendeten Threads wird in `valp` übergeben.

1.4.5 Freigabe einer Thread-Ressource: *pthread_detach()*

Der `wait()`-Ruf für Prozesse hat zwei Aufgaben. Er gestattet dem Aufrufenden sich zu suspendieren, wenn ein Child-Prozeß sich beendet hat. Er erfüllt somit eine wichtige Aufgabe zur Synchronisation. `Wait()` trägt auch die Verantwortung, für die Befreiung des verbleibenden Restes des wartenden Prozesses. Das beinhaltet auch die Kernelaufzeichnung der Existenz der Prozesses.
Pthread_detach(tid) dient dem zweiten Zweck des `Wait()`-Systemrufes. Er verwirft alle Kernelrecords, die zu einem bestehenden Thread gehören. Nach dem Aufruf von *pthread_detach()* , kann auf einen Thread nicht mehr mit *pthread_join()* gewartet werden und der Return-Wert wäre unbrauchbar. *Pthread_detach()* kann auch für einen Thread aufgerufen werden, der noch nicht beendet wurde. In diesem Fall wird der Thread nicht beeinflußt, aber auf ihn kann dann nicht mehr gewartet werden.

1.4.6 Thread Scheduling: pthread_setprio(), pthread_yield()

Threads werden genauso scheduled wie Prozesse. Thread und Prozesse werden über dieselben Prozeßtabellen behandelt. (Stream tasks für die Behandlung von Devices werden außerhalb

dieser Bereiche gescheduled) D.h. die Zustände von Threads/Prozessen werden gemeinsam untersucht, wenn LYNX-OS entscheidet welcher Thread/Prozeß als nächster läuft
Die POSIX- Calls zur Manipulierung der Priorität von Threads sind *pthread_setprio()* und *pthread_getprio()*. Diese können nur innerhalb des Prozesses aufgerufen werden, der den Ziel-Thread enthält.
Wenn ein Thread kreiert wird, dann erhält er eine Priorität, die auf der Attributstruktur, die von *pthread_create* verwendet wird, basiert.
POSIX definiert 2 Arten des Prioritätensetzens.Wenn das pthread_inheritsched- Element in der Attribut- Struktur gesetzt ist, dann erbt der kreierte Thread die Priorität des kreierenden Prozesses. Das ist analog zum Verhaltenvon Prozessen, die fork() aufrufen (Beachte, daß für den Fall, wenn zwei Threads (oder Prozesse im Fall von fork()) die gleiche Priorität haben und der LYNX- Scheduler, ein FIFO, prioritätsbasierender Scheduler ist, daß der neu kreierte Thread (oder. Prozeß) nicht laufen wird, bis der kreierende Thread die Kontrolle freiwillig abgibt (relinguish). Dieser Kommentar gilt nur für die Uniprozessor- Version von Lynx- OS und ist hier angegeben, um an die Art des Lynx-Schedulings zu erinnern.)
Wenn das *pthread_attrib_inheritsched*- Element nicht gesetzt ist, dann erhält der kreierte Thread seine Priorität vom *pthread_attrib_priority*- Element. Damit besteht die Möglichkeit einen Thread zu kreieren, dessen Priorität von der Priorität des kreierenden Prozesses abweicht.
POSIX definiert zwei Calls, die den Prozessor abgeben (relinguish):
POSIX.4 *yield()* arbeitet zwischen Prozessen; POSIX.4a *pthread_yield* zwischen Threads. Tatsächlich sind diese zwei Calls unter LYNX die gleichen, da LYNX Threads und Prozesse identisch scheduled. Aber aus Gründen der Portabilität wird die Verwendung von *pthread_yield()* empfohlen, wenn ein Thread zu einem anderen Thread im selben Prozeß umschalten muß und *yield()* im anderen Fall von Prozessen.
Wenn *pthread_yield* aufgerufen wird, dann wird der rufende Prozeß an das hintere Ende der Warteschlange für sein Prioritätslevel angehängt. Er wird wieder lauffend, wenn er am Kopf der Schlange angelangt ist. Beachte, daß *yield()* einen niederprioriserten Prozeß nicht das Laufen ermöglicht, *yield()* gestattet nur bereiten Prozessen oder Threads auf derselben Prioritätsstufe das Laufen.

1.4.7 Thread Synchronisation: Mutexes, Condition Variables Semaphoren

Da Threads sich ihren gesammten Speicher mit allen anderen Threads im selben Prozeß teilen, gibt es keine natürliche Schutzgrenze, die normaler Weise vorhanden ist beim Programmieren von parallelen Prozessen. Threads werden zwangsläufig kooperierend sein in Real-time - Applikationen. Diese Kooperation behandelt gewöhnlich die Art der Manipulation von geteilten Datenstrukturen. Um zu verhindern, daß mehrere Threads auf gemeinsam genutzte Daten (shared data) in inkorrekter Weise zugreifen, müssen Mechanismen vorhanden sein ,die einen geordneten Zugriff auf solche Daten erlauben.
Es gibt eine große Anzahl von Synchronisationsmechanismen. Binäre Semaphore, zählende Semaphore, Mutexe, Condition Variable, Monitore und das ADA- Rendezvous sind vielleicht die bekanntesten dieser Synchronisationsmechanismen. Beispiele für Synchronisationsmechanismen, die unter Standard- UNIX verfügbar sind, schließen advisory file locking über fcntl() und den Gebrauch von System V- Semaphoren mit ein.
Das Haupthindernis der existierenden UNIX- Synchronisationsmechanismen ist deren Overhead. Jeder existierende Call erfordert wenigstens den Overhead eines Betriebssystem-Calls. Im Gegensatz dazu kann eine einfachere Synchronisation implementiert werden, wenn man im besten Fall zwei oder drei Befehle verwendet oder man einen System-Call verwendet, wenn das Blockieren eines Threads notwendig ist. Beispiele für diese einfachen Synchronisationsmechanismen umfassen Mutexe, Condition Variablen, Semaphoren. Zur Standardisierung in POSIX.4 wurden binäre Semaphoren ausgewählt [P4D9]. In POSIX.4a

wurden Mutexe und Condition Variablen zur Standardisierung gewählt. Getrennt sind die zwei Spezifikationen unvollständig. Zusammen jedoch stellen sie eine komplette Spezifikation für die Synchronisation von parallelen Thread u./o. Prozessen dar. POSIX.4 binäre Semaphore sind gedacht für die Prozeßsynchronisation und sind deshalb langsam und schwerfällig für die Verwendung zwischen Threads. Lynx- Threads stellen zählende Semaphore bereit, die ähnlich den Mutexes und Condition Variablen sind. Diese zählende Semaphore sind wahrscheinlich nützlicher Gebrauch als die POSIX.4 Binärsemaphore bei der Verwendung zwischen Threads.

1.4.8 Verwendung von POSIX- Mutexes

Ein Mutex wird so benannnt, da er den gegenseitigen Ausschluß gewährleistet. Nur ein einziger Thread kann sich in einem Mutex zur Zeit befinden. Sich in einem Mutex befinden bedeutet, sich in einem kritischen Codeabschnitt zu befinden, von dem aus gemeinsam genutzte Datenstrukturen modifiziert werden. Ein Thread ist entweder innerhalb oder außerhalb eines Mutexes. Um in einen Mutex einzutreten ruft ein Thread *pthread_mutex_lock(mut)* auf. Um den Mutex zu verlassen, ruft der Thread *pthrad_mutex_unlock(mut)* auf. Wenn ein Thread versucht in einen Mutex einzutreten, während ein anderer Thread sich in diesem Mutex bereits befindet, dann wird der Thread, der das versucht solange blockiert bis der Mutex „ungelockt“ wird und er dann eintreten kann- Wenn mehrere Threads versuchen in einen Mutex zur selben Zeit einzutreten, dann erhalten sie Zutritt zum Mutex in der Reihenfolge der Prioritäten. Eine alternative Form eines Mutexeintrittes steht zur Verfügung und heißt *pthread_mutex_trylock(mut)*. Das ist ein Versuch den Mutex zu betreten und für den Fall, daß ein Eintritt nicht erfolgen kann, blockiert der Thread nicht. Anstelle dessen gibt *pthread_mutex_trylock(mut)* einen Errorcode zurück, der anzeigt, daß der lock- Versuch blockieren würde. Der Thread der versucht hat den Mutex zu betreten, ist dann frei und kann dann andere Aktionen durchführen.

1.4.9 Verwendung von POSIX- Condition Variblen

Mutexe sind einfache Mechanismen, mit denen sich vielleicht der Großteil der notwendigen Basisisynchronisationen erfüllen läßt. Sie sind aber ungeeignet, um auf ein Ereignis zu warten. Eine weitere Einsatzmöglichkeit eines Mutexes ist der Schutz einer komplexen Datenstruktur. In solch einer Struktur kann ein Thread den Status von Daten abfragen. Dafür ist der gegenseitige Ausschluß erforderlich, um zu sichern, daß ein zusammenhängender „Schnappschuß“ der Datenstruktur vom Thread erkannt wird. Jedoch wenn einmal der Zustand der Datenstruktur abgefragt wurde, dann könnte es sein, daß der Thread auf die Aktualisierung der Datenstruktur warten möchte, um den neuen Status abzufragen. Ein typisches Beispiel ist ein „double buffering algorithm“, bei dem ein Thread für das Auffüllen von zwei Buffern zuständig ist, während ein weiterer Thread verantwortlich dafür ist, zwei Buffer zu leeren. Die Hoffnung hier ist, daß diese beiden Prozesse, zu jeder Zeit auf das Gerät, aus dem gelesen wird und auf das Gerät in das geschrieben wird, zugreifen können, wenn das erforderlich ist.
Ein bedeutender Teil dieses Algorithmusses besteht darin, daß der Thread, der die Buffer auffüllt, dazu fähig sein muß, einen leeren Puffer zu erhalten von dem Thread, der die Puffer leert. Ebenfalls muß der Thread, der die Buffer leert, in der Lage sein, auf einen vollen Buffer zu warten. Mutexe sind dafür nicht geeignet.

1.4.9.1 Warten auf eine Condition Varible

Conditions Variablen sind in enger Verbindung mit einem Mutex als Synchronisationsmechanismus für diese Aufgaben geeignet.
Conditions Variablen werden verwendet, um die Erfüllung von Bedingungen abzuwarten, wie z.B. die Bedingung „der erste Buffer ist jetzt voll“ oder „der zweite Buffer ist jetzt leer“.

Um auf eine besondere Bedingungvariable zu warten, muß ein Thread zuerst den Mutex, der mit der Conditionsvariable verbunden ist, betreten. Das würde gleichbedeutend sein damit, daß der Thread den Zustand der Datenstruktur abfragt. Er betritt den Mutex, um eine zusammenhängende Datenstruktur zu erhalten. Folglich haben alle anderen Threads keinen Zugriff auf diese Datenstruktur. Wenn der Thread feststellen möchte, ob eine Condition eintritt, die mit der Datenstruktur verbunden ist, dann ruft er *pthread_cond_wait(cond,mut)* auf mit der Conditionsvariable und dem Mutex als Argumente.
Pthread_cond_wait() bricht das Mutex ab und reiht den Thread an der Conditionsvariablen nach Priorität geordnet ein und wartet das Signalisieren der Conditionsvariable ab.

1.4.9.2 Signaling Conditions Variable

*Pthread_cond_signal(cond,mut) w*ird verwendet, um einer Conditionsvariblen ein Signal zu senden. Das führt zum Erwachen des höchstpriorisierten Thread, der an der Conditionsvariablen blockiert ist. Der Thread kann den Mutex zu erwerben und mit der seiner Ausführung beginnen. Wenn der signalisierende Thread schon den Mutex genutzt hat, dann kann der wartende Thread nicht sofort mit der Abarbeitung beginnen, wenn er das Signal empfängt. Damit wird verhindert, daß zwei Threads den Mutex zur selben Zeit gelockt haben könnten. Eine Variante von *pthread_cond_signal()* ist *pthread_cond_broadcast()*, welcher alle Threads aufweckt, die an einem Mutex blockiert sind. Mit *pthread_cond_broadcast()* müssen alle diese blockierten Threads den Mutex erneut anfordern, bevor diese die Ausführung fortführen. Das sichert, daß die Threads in serieller Folge mit der höchsten Priorität zuerst aufwachen. Wenn kein Thread an der Conditionsvariable wartet, wenn *pthread_cond_signal()* aufgerufen wird, dann gibt es keinen Effekt.

1.4.9.3 Eine Conditions Variable ist zustandslos

Ein kritischer Punkt ist, wenn ein Thread *pthread_cond_wait()* aufruft, dann wird der rufende Thread immer blockieren, bis ein anderer Thread *pthread_cond_signal(cond)* oder *pthread_cond_broadcast(cond)* augerufen hat. D.h. die Conditions Variable kann sich nicht in einem bereiten- oder unlocked- Zustand befinden, wie es ein Semaphore sein kann. Die Conditionsvariable unterscheidet sich grundsätzlich von einem Semaphor dadurch, daß die Conditions Variable keinen Speicherzustand, der mit ihr verbunden ist. es gibt nur eine Queue von Threads, die auf das Einteten einiger Ereignisse warten. Da Bedingungsvariablen zustandslos sind, muß das Blockieren an einer Condition Variable atomar sein. Das kann mit dem Eintreten in ein Mutex realisiert werden. Das Senden eines Signals an eine Conditionsvariable muß im Zusammenhang mit einem Mutex, das gelockt ist, durchgeführt werden. (Wenn mit Conditionsvariablen gearbeitet wird, dann muß mann auch gleichzeitig mit Mutexen arbeiten.) Ansonsten könnte folgende Wettlaufbedingung auftreten:
Es soll die Annahme gemacht werden, daß eine Applikation eine Datenstruktur verwendet, die nicht durch ein Mutex geschützt ist, aber eine Condition Variable, um auf einige Bedingungen zu warten. Z.B. ist die Bedingung ist TRUE, wenn ein spezielles Bit B gesetzt ist. Ein Thread fragt das Bit ab und stellt fest, daß es FALSE ist. Das bedeutet, daß der Thread blockieren muß, um darauf zu warten, daß B auf TRUE gesetzt wird. Der Thread ist so geschrieben, daß er an einer Condition Variable wartet, aber von einem zweiten Thread unterbrochen wird. Dieser zweite Thread setzt B auf TRUE und für den Fall, daß andere Threads auf dieses Bedingung gewartet haben, ruft der Thread *pthread_cond_signal()* auf, um den wartenden Thread aufzuwecken. Aber es gibt keine wartenden Threads, da der erste Thread nicht blockiert hat an dieser Bedingung. Jetzt scheidet der zweite Thread aus und der erste Thread läuft weiter. Dieser ruft nun *pthread_cond_wait()* und geht dann schlafen. Da die Condition Variable keinen mit ihr verbundenen Zustand besitzt, ist das System nicht in der Lage sich

daran zu erinnern, daß ein Signal schon früher eingetroffen wird. Die Folge ist, daß der erste Thread die Signalisierung des Eintreffens der Bedingung verpaßt hat und folglich für immer schläft. Wenn die Datenstruktur geschützt gewesen wäre durch ein Mutex, dann würde das eben beschriebene Scenario nicht stattfinden.
Um einen Mutex zu kreieren, wird *pthread_mutex_init(mut,attr)* verwendet. Diese Funktion verwendet eine Attributstruktur, die ähnlich der von *pthread_create()* ist. Diese Attributstruktur sollte mit default- Werten unter Verwendung von *pthread_mutexattr_create()* initialisiert werden. Um ein Mutex zu löschen wird *pthread_mutex_destroy(mut)* verwendet. Wie *pthread_cond_init(cond,attr)* eine Bedingungsvariable initilisiert, so intialisiert *pthread_condattr_create(attr)* die Attributstruktur. *Pthread_cond_destroy(cond)* löscht eine Bedingungsvariable.
Warnung: Bedingungsvariablen und Mutexe werden in einer sehr effizienten Form außerhalb des LynxOS 2.0-Kernels implementiert. Aus diesem Grund können sie kein Adreßkontrolle durchführen, wie es echte Systemcalls machen. Wenn man einen der oben genannten Calls mit einem falschen Mutex oder einer falschen Bedingungsvariablen aufruft, dann kann es sehr gut sein, daß die Applikation ein Coredump hinterläßt. Das gleiche gilt bei einem Versuch auf gelöschte Mutexe oder Bedingungsvariablen mittels *lock()* oder *signal()* zuzugreifen.

1.4.10 Verwendung anderer Synchronisationsmechanismen

Mutexe und Bedingungsvariablen sind ein übliches Modell, für die Synchronisation von Threads. Ein anderes gebräuchliches Modell verwendet Semaphore. Ein Semaphor ist verwandt mit einem Mutex

Status	Besitzer	Mechanismus
kein	kein	Konditionsvariable
kein	ja	Kein verfügbarer Mechanismus
binär	kein	Binäres Semaphor
binär	ja	Mutex
zählend	kein	Zählendes Semaphor
zählend	ja	Kein verfügbarer Mechanismus

Tabelle 10.1 Aspekte unterschiedlicher Synchronisationsmechanismen

Ein Semaphor kann ebenso wie ein Mutex frei oder belegt sein. Jedoch kann ein Mutex nur zurückgesetzt werden durch den Besitzer des Mutexes. Das macht den Gebrauch für das generelle Einreihen in Warteschlangen unmöglich (general queuing). Im Gegensatz dazu kann ein Semaphor von jedem Thread gesetzt werden. Folglich ist es wie eine Bedingungsvariable für das Einreihen in Warteschlangen verwendbar.
Was ist nun der Unterschied zwischen einem Semaphor und einer Bedingungsvariablen? Ein Semaphor hat einen Zustand, der mit dem Semaphor verbunden ist und kann sich deshalb erinnern, daß es früher frei bzw. zurückgesetzt war. Folglich wird ein Semaphor ohne ein schützendes Mutex verwendet.
Die POSIX.4 binären Semaphoren können dazu verwendet werden sowohl Threads als auch Prozesse zu synchronisieren. Die zählenden Semaphore der Lynx- Threads sind wahrscheinlich besser zu verwenden, wenn der Gebrauch von Semaphoren notwendig wird. Dieser Semaphorentyp wird im Abschnitt der Lynx- Threads beschrieben. Die Tabelle gibt an welche Synchronisation welche Attribute besitzt.

1.4.11 Schnelles Beenden anderer Threads: *pthread_cancel()*

Wenn es notwendig ist, einen anderen Thread zwangsweise (forcibly) zu beenden, dann kann ein anderer Thread *pthread_cancel(tid)* aufrufen. *Pthread_cancel()* bewirkt, daß der Zielthread wie beim Ruf *pthread_exit()* beendet wird. Zusätzlich dazu wird jeder Cleanup- Handler (s. weiter unten) aufgerufen, der durch den Opferthread gesetzt wurde, um die Arbeiten zu beenden, die der Opferthread durchgeführt hat.
Der Opferthread kontrolliert, ob er gecancelt werden kan oder nicht. Zusätzlich dazu kann der Opferthread sein Beendigungsstatus setzen, um zu bestimmen, unter welcher Bedingung er beendet wird. Insbesodere bedeutet das, daß ein Thread nur dann gelöscht werden kann, wenn der Opferthread, das dem löschenden Thread erlaubt hat. Der Opferthread kann seinen Beendigungsstatus modifizieren, so daß der Beendigungsbefehl keinen Effekt hat. Wenn ein Thread *pthread_setgencancel(flag)* aufruft, dann kann er seinen generellen Beendigungsfähigkeitsstatus (generell cancellability) ein- oder ausschalten, in Abhängigkeit von dem dort angegebenen Flag. Falls die Möglichkeit, den Thread zu canceln, ausgeschaltet ist, dann wird das Löschen des Threads, der dieses nicht erlaubt hat, zum Blockieren führen, ähnlich einem blockierenden Signal. Falls die generelle Beendigungsfähigkeit eingeschaltet ist, dann sind Beendigungsanforderungen erlaubt und können ausgeführt werden.
Weiter bestimmt der asynchrone Beendigungsfähigkeitstatus, wenn „general cancellability“ gesetzt ist, ob die Beendigungsanforderung gleich nach dem *pthread_cancel()* durchgeführt wird (d.h. asynchron) oder erst an einem spezifischen synchronen Endpunkt. Asynchrone Beendigung wird mit *pthread_setasynccancel(flag)* gesetzt.
Falls die generelle Beendigungsfähigkeit eingeschaltet ist und die asynchrone Beendigungsfähigkeit ausgeschaltet ist, dann kann die Beendigung nur an Beendigungscheckpunkten vorkommen. Das sind:

1. Ein Ruf zu *pthread_testcancel()* durch den Opfer-thread.
2. Ein Ruf zu *pthread_cond_wait()* und seinen Varianten durch den Opferthread.
3. Ein Ruf zu *pthread_join()* durch den Opferthread.

Falls die generelle und asynchrone Beendigungsfähigkeit eingeschaltet sind, dann kann der Thread zu jeder Zeit beendet werden.
Die Verwendung von *pthread_cancel()* geht nicht mit dem synchroner Modell der Thread-Programmierung konform und deshalb wird empfohlen, daß *pthread_cancel()* nur bei einem unerwarteten Ereignis oder in einer Gefahrensituation zu verwenden ist.

general cancellability	asynchronous cancellability	
off		führt zum Blockieren
on		Abbrechen eines anderen Threads erlaubt
on	on	Abbruch direkt hinter pthread_cancel. Thread kann kann zu jeder beliebigen Zeit abgebrochen werden.
on	off	Abbruch nur an einem zulässigen Abbruchpunkt. Aufruf von: • *pthread_testcancel()* • *pthread_cond_wait()* + Varianten • *pthread_join()* durch Opferthread

1.4.12 Die Bereinigungsfähigkeit eines Threads beim Terminieren: pthread_cleanup_push

Jeder Thread enthält einen Stack von Cleanup-Handlern, die *pthread_cleanup_push(routine,arg)* und *pthread_cleanup_pop(execute)* verwenden. Wenn der Thread abgebrochen wird und die Abbruchanforderung Schwierigkeiten bereitet (s. Diskussion oben) oder wenn der Thread *pthread_exit()* aufruft oder wenn der Thread von seiner Initialisierungsroutine zurückkehrt, dann wird jede Cleanup- Handler- Routine aus dem Stack herausgeholt und mit seinem Argument *arg* ausgeführt. Das erlaubt einem Thread seine Dinge zu ordnen, bevor er weiterläuft. Threads können diese Handler selbst unter Verwendung von *pthread_cleanup_pop(execute)* aufrufen. Wenn *execute* wahr ist, dann wird die Routine ausgeführt.

1.4.13 Thread-spezifische Daten: pthread_keycreate(),pthread_setspecific()

Threads können sich bei Verwendung Thread-spezifischer Daten eine Privatspäre schaffen. Auf Thread-spezifische Daten kann mittels eines Schlüsselwertes zugegriffen werden. Thread-spezifische Daten unterscheiden sich für jeden Thread im System. Um einen neuen Schlüssel zuzuweisen, wird *pthread_keycreate(keyp, destruktor)* verwendet. Diese Funktion weist eine neuen Schlüssel zu und verbindet die Destruktorfunktion mit diesem Schlüssel. Nachfolgende *pthread_setspecific()*- Rufe rufen die Destruktorfunktion mit dem alten Wert der threadspezifischen Daten auf. *Pthread_setspecific(key,val)* und *pthread_getspecific(key)* setzen bzw. liefern die Datenwerte für den Thread. Der Datenwert für den jeweiligen Thread umfaßt die Größe eines Pointers, die es ermöglicht durch einen Pointer auf eine größere Daten-Struktur zuzugreifen.
Der Destruktor wird beim ersten Aufruf von *pthread_setspecific()* durch einen besonderen Thread mit einem besonderen Schlüssel nicht aufgerufen, da beim ersten Mal noch kein alter Wert existiert, mit dem die Destruktorfunktion aufgerufen wird.

1.4.13.1 Errno ist Thread-spezifisch

Ein Wert von Thread- Daten ist im System inbegriffen: errno. Falls ein Thread einen Fehler als Ergebnis eines Systemrufes erhält, dann kann die Fehlernummer nur vom Thread gesehen werden.

1.4.14 Die Thread nur- einmal- Initialisierungseigenschaft: pthread_once()

Globale Einrichtungen in Programmen erfordern oft eine Initialisierung, die nur einmal durchlaufen wird. Die normale Vorgehensweise eine Initialisierung durchzuführen, zeigt der folgende Code:

```
static int package_initialized = 0;
initialized_package()
   {
if (!package_initialized(){
package_initilized = 1;
/* Initialize package() */
        }
    }
```

Dieser Code ist unangemessen in einer multithreaded Applikation, da eine Wettlaufbedingung zwischen zwei Threads stattfinden kann, um den Initialisierungscode auszuführen. Zwei

Threads könnten gleichzeitig den Wert der statischen Variablen testen; und falls beide den Wert gleich Null finden, dann führen beide Threads diese Initialisierung durch. Um diese Art von Aktionen zu verhindern, wird eine atomare Test- und Set- Operation gebraucht. Weiterhin ist eine Blockieren erforderlich, um weitere Fehler zu verhindern, die auftreten können, wenn man mehrereThreads verwendet.
Durch den Aufruf von *pthread_once(once_only, routine)* wird es einem Thread bei der Rückkehr von diesem Call garantiert, daß die Initialisierungscoderoutine nur einemal aufgerufen wurde. Der once_only- Block steuert den Zugriff auf diese Routine und er muß initialisiert werden auf den statischen Initialisierungswert `pthread_once_init`:

```
pthread_once_t module_initialized = pthread_once_init;
```

1.4.15 Wiedereintrittsfähige Threadfunktionen:

Einige existierende Bibliotheksroutinen sind nicht sicher, um durch mehere Threads gleichzeitig aufgerufen zu weden, da sie Zwischenergebnisse in einem globalen Puffer entweder während oder zwischen den Funktionsaufrufen speichern. Auf den globalen Puffern könnte durch mehrere Threads zur gleichen Zeit zugegriffen werden. Das führt zu Fehlern. POSIX enthält eine Anzahl von Funktionen in einer Thread- sicheren Form, die User- verbundene Puffer verwendet und keine globalen Puffer. Diese neugeschriebenen Routinen sind verschieden von den Originalroutinen, da sie mehr Argumente enthalten. Die zusätzlichen Argumente sind Puffer, die entweder als „scratchpads“ oder als „return buffers“ verwendet werden von der rufenden Funktion. Im allgemeinen heißt die wiedereintrittsfähige Threadfunktion do_something_r, wobei „r“ für reentrant steht. Funktionen, die die Systemdatenbasen manipulieren liegen in der wiedereintrittsfähigen Version vor: getpwnam_r, getpwuid_r, getpdid_r usw. Ebenso liegen die Zahlenmanipulationsfunktionen in der wiedereintrittsfähigen Version vor: gmtime_r usw. Eine komplette Liste der wiedereintrittsfähigen Threadfunktion , wird in einem anderen Abschnitt gegeben.

1.4.16 Nichtwiedereintrittsfähige Threadfunktionen

Standardfunktionen wurden neugeschrieben, um für mehrere Threads sicher zu sein, die diese Standardfunktion gleichzeitig aufrufen. Das erfordert im allgemeinen keinen Wechsel des Interfaces wie in den darüberliegenden Beispiele, deshalb ist es transparent für den Programmierer. Jedoch erfordert multithreading zusätzlichen Synchronisationsoverhead. Die Funktionen `malloc` und `printf` sind die beiden längsten Standardfunktionen, die für den Gebrauch in multithreaded- Anwendungen sicher sein müssen. Für jede dieser Fähigkeiten wurden bei den kritischen Abschnitten Mutexe hinzugefügt, um zu verhindern, daß Threads auf gemeinsam genutzte Puffer konkurierend zugreifen.
Der Synchronisationsoverhead kann in vielen Applikationen unnötig sein. Für die Familie der printf- Routinen putc, puts, getchar, fprintf, fscanf, flushbuf, fopen, fclose, gibt es Versionen, die nicht Thread- sicher sind, die aber besser ausgeführt werden können, falls der gemeinsame Zugriff nicht erforderlich ist. Diese Routinen heißen unlocked_printf, unlocked_flushbuf usw.
Manchmal kann es in Applikationen mit mehreren Threads effizienter sein, den gegenseitigen Ausschluß über ein File zu realisieren. In solchen Situationen sollten Threads einen exklusiven Zugriff auf einen Filepointer (FILE) mit lockfile(file) haben. Dann können beide Versionen, die unlocked oder die multithreaded, ohnen Schaden anzurichten von Threads verwendet werden. Der Thread sollte den gegenseitigen Ausschluß für das File unter Verwendung von funlockfile(file) freigeben. wenn der Thread sein I/O zum File macht.

Standard Funktion	Thread-Reentrant Version
opendir(dirname)	opendir_r(dirname, dirp)
readdir(dirp)	readdir_r(dirp, direntp)
localtime(clockp)	localtime_r(resultp, clockp)
gmtime(clockp)	gmtime_r(resultp, clockp)
asctime(timep)	asctime_r(timep, bufp, len)
ctime(clockp)	ctime_r(clockp, bufp, len)
ctermid(bufp)	ctermid_r(bufp)
getlogin()	getlogin_r(bufp, len)
ttyname(fd)	ttyname_r(fd, bufp, len)
getpwnam(name)	getpwnam_r(passwdp, name, bufp, len)
getpwuid(uid)	getpwuid_r(passwd, uid, bufp, len)
getgrname(name)	getgrname_r(group, name, bufp, len)
getgrid(uid)	getgrid_r(group, uid, bufp, len)
strtok(s1,s2)	strtok_r(s1,s2, lasts)

Multi-Thread-Safe Function	Unlocked Version
getc(stream)	unlocked_getc(stream)
fgetc(stream)	unlocked_fgetc(stream)
getw(stream)	unlocked_getw(stream)
getchar()	unlocked_getchar()
putc(c, stream)	unlocked_putc(c, stream)
fputc(c,stream)	unlocked_fputc(c,stream)
putw(w, stream)	unlocked_putw(w, stream)
putchar©	unlocked_putchar©
printf(fmt,...)	unlocked_printf(fmt,...)
fprintf(stream,fmt,...)	unlocked_fprintf(stream,fmt,...)
scanf(fmt,...)	unlocked_scanf(fmt,...)
fscanf(stream, fmt,...)	unlocked_fscanf(stream, fmt,...)

1.5 Der Einfluß von mehreren Threads auf die UNIX- Semantik

Das Vorhandensein von Threads hat Einfluß auf die Arbeitsweise einer Anzahl von existierenden UNIX-Call's. Dieser Abschnitt diskutiert die Auswirkungen von Threads auf existierende UNIX-Systemeigenschaften.

1.5.1 Intervall Timer und SIGALRM

Der Real-Time- Timer, der unter Verwendung von *setitimer(ITIMER_REAL...)* von einem Prozeß manipuliert wird, ist ein „**per-thread entity**“ Das erlaubt mehreren Threads in einem Prozeß in einer unabhängigen zeitgesteuerten Art zu operieren, ohne auf einen Timer auf Applikationsniveau umzuschalten, daß bedeutet daß SIGALRM auf einer per-thread Basis ausgesandt wird

1.5.2 Fork von einem Multi-threaded Programm

Der *fork()*-Aufruf dupliziert den aufrufenden Prozeß. Wenn dieser ein Single-Thread Prozeß ist, dann ist sein Verhalten einfach. Der neue Prozeß ist ein Duplikat des ausgeführte Single-Thread Prozesses. Aber wenn der zu erzeugte Prozeß ein Multi-Thread-Prozeß ist, dann ist die Anzahl der Threads in dem neuen Prozeß nicht sofort klar. Unter POSIX, und damit auch unter LynxOS, hat der neue Prozeß nur einen Thread in sich, das ist ein Abbild des Threads, der den fork-Aufruf ausführte. Warum ist das so? Man stelle sich vor, ein Programm enthält

drei Threads. Einer ruft *fork()* auf, ein anderer mit geringerer Priorität rechnet und ein dritter ist blockiert in einen *read()* Aufruf. Es ist einleuchtend was mit dem *fork()*-aufrufenden Thread passiert. Es wird ein ablauffähiges Abbild von diesem Thread erzeugt. Diese Semantik ist völlig sauber. Dieses gilt auch für den zweiten Thread, welcher nicht im Dialog mit seiner Umwelt steht. Aber was ist mit dem dritten Thread? Der *read* verursacht eine I/O Aktivität, vielleicht DMA, für den Prozeß. Es ist unpraktisch das I/O für die beiden Prozesse zu duplizieren. Gleichfalls ist es unmöglich bei einer blockierenden Operation, zwei Blockierungen zu erzeugen. Die Semantik dieses Threads wäre nur schmerzlich zu ertragen. Es ist einfacher zu erklären, daß der neu kreierte Prozeß nur einen einzelnen Thread hat.
Aus diesem Verhalten können Probleme resultieren. Andere Threads in einem Multi-Thread Programm können einen Riegel vor gemeinsam genutzte Datenstrukturen (shared data structures) im alte Prozeßabbild setzen**.** Dieser Riegel würde in einem neuen Prozeßabbild dupliziert werden und wäre die Ursache dafür, daß der Single-Thread im neuen Prozeß blockieren würde, wenn der Thread auf diesen Riegel auflaufen würde. Ein gutes Beispiel ist die Multi-Thread Version von *printf().*
Die Multi-Thread-*printf()*-Funktion wurde so erweitert, daß, wenn mehrere Threads diese gleichzeitig aufrufen, sie in einer Warteschlange eingereiht werden und somit gesichert wird, daß all Rufe nacheinander ausgeführt werden. Dieses verhindert eine mögliche zerstückelte Ausgabe, wenn zwei Threads auf einmal versuchen, printf auszuführen.
Die Art und Weise wie das getan wird, geht von der Verwendung einer Variante des MUTEXes (s.o.) aus, um einen wechselseitigen Ausschluß für jeden Puffer, der für I/O verwendet wird, zu gewährleisten. Die Mutexes werden belegt und freigegeben; im Standardfall durch das Setzen und Rücksetzen vom Bits im Usermemory.
Wenn einer der Threads den Mutex betreten (locks) hat, und ein andere Thread fork() aufruft, dann erhält der neue Prozeß ein Image des aktuellen printf-Puffer's- einschließlich des blockierten Mutexes und des teilweise abgeschlossenen I/Os. Wie man sieht, gibt es mit jeder Art von *fork()* Probleme. Für die richtige Nutzung von fork() wird empfohlen nur eine Funktion der *exec()*-Familie nach einem *fork()* aufzurufen oder für eine Art geordnetes Runterfahren zu sorgen, um alle Threads zu beenden, bevor fork() aufgerufen wird. Dieses ist das übliche Verhalten. Fork-Aufrufe sind relativ selten.

1.5.3 Exec von einem Multi-Threaded Programm

Exec und seine verwandten Funktionen, haben einfache Ergebnisse, wenn ein Thread in einem Multi-Thread-Programm aufgerufen wird. Exec ersetzt das aktuelle Prozeßimage mit dem Image des neuen Prozesses, und beginnt mit der Ausführung des Prozesses am Eintrittspunkt (i.d.R `main`). Im neuen Prozeß, kann nur ein Thread existieren, da nur ein Eintittspumkt existiert, und es ist sicher, daß keine Anweisung mehr von den alten Threads laufen- Die Anweisungen des alten Prozesses wurden überschrieben vom neuen Prozeßimage. Folglich löscht *exec* alle Threads in dem Prozeß, bevor das Prozeßimage überlagert wird und das neue Programm gestartet wird.

1.5.4 Exit und Core Files

Exit in einem multithreaded Prozeß löscht alle Threads in diesem Prozeß. Ebenso bewirken Signale, die ein Löschen des Prozesses zur Folge haben auch ein Löschen aller Threads in diesem Prozeß.
Wenn eine Signal die Erzeugung eines Core-Files zur Folge hat, dann stoppt der Kernel alle Threads im Prozeß, speichert deren aktuellen Zustand im Core File, löscht dann die Threads und endet. Das genaue Format eines Core Files ist in der Header-Datei core.h zu finden.

1.5.5 Threads und Signale

Die Nutzung von UNIX Signalen wird beeinflußt durch das Vorhandensein von Threads. Threads erleichtern das Schreiben von einfachem synchronen Code, der Signale bedient. „Synchroner“ Code bezieht sich auf Code, der sich beim Signalaufruf nicht um asynchrone Interrupts zu kümmern braucht.
Ohne Threads hat man sich gewöhnlich um ein Signal zu kümmern, das an jedem beliebigen Punkt innerhalb einer Prozeßausführung eintrifft. Dann muß man Vorkehrungen treffen, um die Ausführung des Signalhandlers zu verhindern und damit ein Zerstören von Datenstrukturen vorzubeugen, die sich in einem inkonsistenten Zustand befanden, als das Signal auftrat. Wenn Threads verwendet werden, kann man einfach einen Thread erzeugen, der auf ein Signal wartet und eine Aktion ausführt, in Abhängigkeit von dem eintreffendem Signal. Das befreit die anderen Threads der Applikation von der Behandlung von Signalen, sofern diese von Threads bedient werden.
Jedoch bringen Threads auch zusätzliche Komplexität. Wenn sich in einem Prozeß viele Threads befinden und ein Signal an diesen Prozeß gesandt wird ergibt sich folgende Frage: Welcher Thread bekommt das Signal?
Um diese Frage beantworten zu können, teilt man Signale in zwei Klassen: die Pro-Thread Signale und die Pro-Prozeß Signale. UNIX-Signale werden für zwei verschiedene Dinge gebraucht: das Exception Handling und das Interrupt Handling. Exception-Handling muß durchgeführt werden, wenn eine Bedingung, wie z:B. eine illegaler Befehl, oder ein Speicherfehler als ein direktes Ergebnis dessen, was ein Thread tat, auftritt. Interrupt Handling kommt asynchron vor, und hat mit dem Code des Threads nichts zu tun, den dieser gerade ausführt

1.5.5.1 Thread Signale

Die threadspezifischen Signale sind Exceptionhandling-Signale und resultieren aus dem Verhalten der Threads. Mit anderen Worten, wenn ein Thread einen Speicherfehler verursacht, dann erhält dieser und kein anderer Thread das entsprechendes Signal, das mit dem Fehler korrespondiert. Thread Signale sind SIGSEGV, SIGBUS, SIGTRAP, SIGALRM und SIGFPE.
SIGALRM ist das Resultat eines Zeitüberschreitung, und kann deshalb als asynchron betrachtet werden. Timer sind threadspezifisch: Jeder Thread hat seinen eigenen Intervalltimer. Dieser ist notwendig, um dem Thread zu signalisieren, das die Zeit abgelaufen ist. Es ist notwendig, daß für den Fall, daß das Zeitintervall für den Thread abgelaufen ist, auch dieser Thread das Signal erhält und kein anderer.
Wenn ein Thread ein Signal maskiert, blockiert, ignoriert oder behandelt, hat das keine Auswirkungen auf das Maskieren, Blockieren, Ignorieren und Behandeln der gleichen Signale für die anderen Threads im Prozeß.

1.5.5.2 Prozeßspezifische Signale

Alle anderen Signale sind Signale für Prozesse. D.h., daß das System eine Entscheidungsprozedur durchläuft, um zu bestimmen, welcher Thread im Prozeß das Prozeß-Signal erhält. Das bedeutet weiter, wenn ein Threads in solches Signal blockiert, ignoriert, maskiert oder behandelt, dann gilt dies auch für alle anderen Threads im Prozeß. D.h.,daß Signalhandler werden auch auf der Basis von Per-Thread-Signalen vom System unterstützt, aber die Maskierung von Signalen bezieht sich auf den Prozeß.

Beachte:

Standardsignalaktionen sind prozeßweite Aktionen. Zum Beispiel, wird beim Auftreten des Signals SIGSEGV der Prozeß beendet und ein Core-File im aktuellen Verzeichnis hinterlassen. Wenn das Signal SIGSEGV an einen bestimmten Thread gesandt wird, und der Thread enthält die Standardreaktion, dann wird der gesamte Prozeß beendet und somit auch alle im Prozeß existierenden Threads. Das ist ein kleiner , aber wichtiger Punkt, den man im Auge behalten sollte.
Es folgt das Entscheidungsverfahren nach welchem entschieden wird, welcher Thread ein Pro-Prozeß Signal erhält.

1. Wenn ein Thread *sigwait()* für ein Signal verwendet, dann bekommt dieser Thread das Signal. Wenn aber mehr als ein Thread mit *sigwait()* auf das selbe Signal warten, dann wird ein Thread zufällig ausgewählt, um das Signal zuerhalten.
2. Wenn das Signal ignoriert wird, dann wird es übergangen. Beachte, *sigwait()* übergeht das Ignorieren eines Signals!
3. Wenn ein Signal blockiert ist, bleibt es aber für den Prozeß erhalten. Wird die Blockierung aufgehoben, dann wird das Signal dem Thread (wie unter Viertens) zugestellt. (Ebenso, wenn ein Thread später eingeführt wird und das Signal anfordert, wird er das Signal sofort erhalten)
4. Sonst wird ein Thread zufällig ausgewählt und das Signal diesem Thread dann zugestellt. Das Signal wird nicht an einen bereits gelöschten Thread weitergeleitet. Bei einen solche Versucht, wird das Signal auf die bekannte Weise zugestellt. Wenn ein bestimmtes Thread ein Signal bekommen soll, sollte man diese Art der Zustellung vermeiden.

1.5.5.3 Taktik für die Verwendung von Threads und Signalen: Ein Travel Advisory

Nicht immer ist es von Vorteil alle Möglichkeiten die einen das System gibt auch zu nutzen. Im speziellen weisen wir darauf hin, daß beim Thread-Ansatz von POSIX.4a noch radikale Änderungen, auf dem Gebiet der Signalbearbeitung, auftreten können. Im Augenblick ist es schwierig zu sagen, wie am Schluß die gesamte Semantik von Threads und Signalen aussehen wird. Wie auch immer, es gibt in der POSIX-Gruppe eine Übereinkunft über den "richtigen Weg", wie Signale in einer Thread-Anwendung zu nutzen sind. Der "richtige Weg" sollte effizienter sein, und eine bessere Wartung und lesbarkeit realisieren. Wir raten deshalb dringend, daß Anwendungen, die Threads nutzen, auf folgende weise mit Signalen umgehen sollten:

- Pro-Thread Signale: Es sollten identische Thread-Signalhandler für jeden Thread realisiert werden. Das wird im allgemeinen der Normalzustand sein: jeder Thread wird auf identische mit SIGSEGV umgehen, paramertersiert nur durch die Art und Weise wie das Signal vorkommt. Threads dürfen Signale auf der per-thread- Basis noch blockieren, aber der Programmierer sollte die Handler als potentielle Prozeßeintritte betrachten.
- Pro-Prozeß Signale: Zum Blockieren aller Pro-Prozeß Signale nutzt man *sigsetmask()*. Verwende zugehörige Threads, um die gesamten Pro-Prozeß Signale mit *sigwait()* zu bedienen. *Sigwait()* unterstützt ein sichereres und effizienteres Signalhandling für die Prozeß-Signale als eine asynchrone Signalbehandlung. Vermeide die Signalbehandlung für Prozeß Signale. Erstens ändert sich wahrscheinlich die Semantik von diesen Handlern, und zweitens machen Threads asynchrone Signalbehandlungen unnötig. Die Nutzung von Threads für synchrone Signale ist eine viel effizientere, robustere und flexiblere Methode des Umgangs mit diesen Signalen.

Mit anderen Worten, man behandelt alle Pro-Thread Signale in der gleichen Art und Weise, und bedient die gesamten Pro-Prozeß Signale durch Nutzung von *sigwait()*. Die asynchrone Semantik der Signale wird wahrscheinlich verändert bevor POSIX.4a abgeschlossen ist. Man kann aber davon ausgehen, daß die Elemente die gleichen bleiben werden.

1.6 Anhang: Lynx Threads gegenüber POSIX Threads

Um die Eigenschaften der Lynx- Threads zu verwenden, muß das Headerfile <st.h> eingebunden werden. Die -m Option muß ebenfalls beim Compiler angegeben werden. Lynx-Threads u. POSIX- Threads sind entsprechend dem POSIX- Interface gleich. Jedoch sind die Unterschiede groß genug, so daß ein Mix aus beiden Interfacen katastropal enden würde. Aus diesem Grund sollte man nur die Lynx- Thread- Eigenschaften verwenden für die es keine Analogie zu den POSIX- Threads gibt. Diese werden jetzt beschrieben.

1.6.1 Das Stoppen von Threads

POSIX 1003.4a spezifiziert keine Methode einen Thread anzuhalten. Die Lynx **st-** Funktionen unterstützen Funktionen zum Stoppen und Beenden eines anderen Threads und folglich auch zum wiederstarten. Um einen Thread zu stoppen verwendet man *st_stop (tid).* Um die Ausführung eines Threads erneut zu starten, wird *st_resume(tid)* verwendet. Diese Rufe manipulieren einen Zähler, der für jeden Prozeß vorhanden ist. Folglich inkrementieren *st_resume()*- Rufe und dekrementieren st_stop()- Rufe diesen Zähler. Ein Thread wird nur gestoppt, wenn der Zähler Null oder neagtiv wird. Ein gestoppter Thread kann nur wiedergestartet werden, wenn der Zähler positiv ist.

1.6.2 Das Kreieren von Threads

Lynx sts können mit *st_new(routine,arc,attrp,tidp)* kreiert werden. Diese Funktion ist analog zu *pthread_create().* Jedoch hat die Attributstruktur das zusätzliches Element `st_state`. Dieses Feld indiziert, ob der neu kreierte Thread im gestoppten Status kreiert wird oder nicht. Wenn dieses Feld Null ist, dann wird der Thread mit einem Startwert von Null kreiert und ist somit nicht lauffähig, wenn der Zähler positiv wird. Im anderen Fall wird der Thread mit einem Startwert von eins kreiert und kann ablaufen. Alternativ dazu kann die primitive Routine *st_build()* verwendet werden. *St_build (routine, arg, stacksize, priority)* kreiert immer einen gestoppten Thread mit angegebenen Priorität.

1.6.3 Zählende Semaphore

Mit den Lynx- Threads werden zählende Semaphore zur Verfügung gestellt. Die zählenden Semaphoren sind in ihrer Handhabung einfacher zu verwenden in Verbindung mit POSIX-Threads. Jedoch unterscheidet sich das Interface der lynxeigenen zählenden Semaphoren von den POSIX- Mutexen und Conditionsvariablen. Ein zählendes Semaphor kann gesetzt (signaled) und rückgesetzt (waited upon) werden. Sie werden verwendet indem man *csem_signal()* und *csem_wait().* Wenn ein Thread *csem_wait()* für ein zählendes Semaphor ruft und dieses bewirkt, daß der Semaphor einen negativen Wert erhält, dann wird der Thread in einer Warteschlange entsprechend der Prioritätsordnung eingereiht. Wenn ein Thread *csem_signal()* zu einem zählenden Semaphor ruft, dessen Wert negativ ist, dann wird der höchstpriore wartende Thread erweckt. Wenn der Wert Null ist oder negativ, dann wird Wert einfach inkrementiert. Folglich „erinnern“ sich die zählenden Semaphoren, wie oft sie signalisert wurden.

1.6.4 Das Kreieren eines zählenden Semaphors

Zum Kreieren eines zählenden Semaphores wird *csem_create()* verwendet. Der Ergebniswert dieser Funktion ist ein Deskriptor des zählenden Semaphors. Dies ist anders als bei den Initialisierungsfunktionen eines POSIX- Mutexes oder einer Conditionsvariablen. Die POSIX-Funktionen initialisieren einen Mutex im Speicher, wo der Mutex abgelegt ist; die Lynx-Funktionen allozieren ein neues zählendes Semaphore und geben einen Pointer auf dieses zurück.

1.6.5 Das Löschen eines zählenden Semaphors

Zum Löschen eines zählenden Semaphores wird *csem_delete()* verwendet. Diese Funktion gibt den Speicherplatz, der von dem zählenden Semaphor verwendet wurde, wieder frei. Wenn man auf ein gelöschtes zählendes Semaphor zugreifen möchte, erhält man ein „Segmentation fault“ als Resultat.

1.6.6 POSIX- Thread- Calls und deren Lynx- Thread- Äquivalente

Folgende Tabelle zeigt die proprietären Lynx- Analogien für die gegebenen POSIX-Funktionen. In einigen Fällen erkennt man, daß die Lynx- Eigenschaften öfter flexibler und leistungsfähiger sind und vielleicht besser in Applikationen geeignet sind.

POSIX Thread Call	Lynx Thread Call
pthread_attr_create(a)	st_default_attr(a)
pthread_attr_delete(a)	keine Lynx- Thread- Analogie
pthread_create(t,a,s,x)	st_new(s,x,a,t)
pthread_join(t,s)	st_join(t,s)
pthread_detach(t)	st_detach(t)
pthread_exit(s)	st_exit(s)
pthread_self()	getstid()
pthread_equal(t1,t2)	keine Lynx- Thread- Analogie
keine POSIX- Thread- Analogie	st_build(r,stk,pr,a,s)
keine POSIX- Thread- Analogie	st_stop(tid)
keine POSIX- Thread- Analogie	st_resume(tid)
pthread_setprio(t,p)	setpriority(0,BUILDPID(0,(t)), (p))
pthread_getprio(t)	getpriority(0,BUILDPID(0,(t)))
pthread_setscheduler(t,s,p)	setscheduler(BUILDPID(0,(t),(s), (p)))
pthread_getscheduler(t)	getscheduler(BULIDOID(0,(t)))
pthread_yield()	yield()
pthread_mutexattr_create(a)	keine Lynx- Thread- Analogie
pthread_mutexattr_delete(a)	keine Lynx- Thread- Analogie
pthread_condattr_create(a)	keine Lynx- Thread- Analogie
pthread_condattr_delete(a)	keine Lynx- Thread- Analogie
pthread_mutex_init(m,a)	mutex_create()
pthread_mutex_destroy(m)	mutex_delete()
pthread_cond_init(c,a)	cv_create()
pthread_cond_destroy(m)	cv_delete()
keine POSIX- Thread- Analogie	csem_create()
keine POSIX- Thread- Analogie	csem(create_val()
keine POSIX- Thread- Analogie	csem_delete()
pthread_mutex_lock(m)	mutex_enter(m, NULL)
pthread_mutex_trylock(m)	mutex_enter(m, _zero_timeout)
pthread_mutex_unlock(m)	mutex_exit(m)
pthread_cond_wait(c,m)	cv_wait(c,m,NULL)
pthread_cond_timedwait(c,m,t)	cv_wait(c,m,t)
pthread_cond_signal(c)	cv_signal(c)
pthread_cond_broadcast(c)	cv_broadcast(c)
keine POSIX- Thread- Analogie	csem_wait(c)
keine POSIX- Thread- Analogie	csem_signal(c)
pthread_cleanup_push(r,a)	st_pushrecover((r),(a))
pthread_cleanup_pop(x)	st_poprecover(x)
pthread_set_cancel(s)	st_setgencancel(s)
pthread_set_asynccancel(s)	st_setasynccancel()
pthread_test_cancel()	st_testcancel()
pthread_cancel(t)	st_cancel(t)
pthread_keycreate(k,d)	st_keycreate((d),(k))
pthread_setspecific(k,v)	st_setspecific((k),(v))
pthread_getspecific(k,v)	st_getspecific((k),(v))

BIBLIOGRAPHIE

Literatur:

[1] IEEE. POSIX 1003.4 Draft 9

[2] IEEE. POSIX 1003.4a Draft 4