Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Informatiker

Thread Debug Interface - TDI

Eine einheitliche Thread-Debug-Schnittstelle für Implementierungen des POSIX-Threads-Standards 1003.1c

Daniel Schulz
Matrikel 111033
Lehrstuhl für Softwaretechnik und Theorie der Programmierung II
Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät II
Humboldt-Universität zu Berlin

Berlin, den 29. Juli 1999

Gutachter:
Prof. Dr. sc. nat. Klaus Bothe
Dr. Frank Müller

Für Claire, Edeltraud und Peter.

Eigenständigkeitserklärung

Hiermit erkläre ich, daß die vorliegende Arbeit

„TDI - Eine einheitliche Thread-Debug-Schnittstelle für Implementierungen des POSIX-Threads-Standards 1003.1c“

von mir selbst und ohne jede unerlaubte Hilfe nur unter Verwendung der angegebenen Quellen angefertigt wurde.

Berlin, den 29. Juli 1999 Daniel Schulz

Einverständniserklärung

Hiermit erkläre ich mein Einverständnis, daß diese Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin öffentlich ausgelegt wird.

Berlin, den 29. Juli 1999 Daniel Schulz

1. Thesen zur Diplomarbeit

Threads eignen sich für die Ausführung von potentiell parallel ausführbaren Aufgaben, die Bearbeitung von sich überlappenden Ein- und Ausgabeoperationen und die Behandlung asynchroner Ereignisse. Sie bieten die Möglichkeit des Echtzeitschedulings und der schnellen Intra-Prozeß-Kommunikation über gemeinsam genutzten Speicher.

Die Standardisierung einer C-Schnittstelle für die nebenläufige Programmierung durch das POSIX-Gremium stellt die Basis für die Entwicklung portabler nebenläufiger beziehungsweise paralleler Multithread-Anwendungen dar.

Dem positiven Aspekt der Entwicklung portabler Multithread-Anwendungen steht jedoch bisher die ungenügende und zudem uneinheitliche Werkzeugunterstützung für die Fehleranalyse und -behebung gegenüber.

Das Debugging von Multithread-Anwendungen gestaltet sich sehr unkomfortabel, wenn der verwendete Debugger die einzelnen Kontrollflüsse nicht erkennt. Für ein Thread-Debugging ist es erforderlich, einzelne Kontrollflüsse zu selektieren und Befehle an einzelne Threads zu binden.

Die wenigen Vertreter der Thread-Debugger, mit denen sich Multithread-Anwendungen debuggen lassen, sind an spezielle Pthreads-Implementierungen und Plattformen gebunden und damit nicht für beliebige standardkonforme Pthreads-Implementierungen benutzbar.

Zusätzlich zum API werden im POSIX-Threads-Standard explizit und implizit Definitionen der Objekte Thread, Mutex und Bedingungsvariable sowie ihrer Attribute vorgenommen. Damit läßt sich ein einheitliches Datenmodell für alle POSIX-Threads-Laufzeitumgebungen erstellen, das die Entitäten Thread, Mutex und Bedingungsvariable enthält.

Das TDI (*Thread Debug Interface*) soll die Möglichkeit bieten, die Attribute der genannten Entitäten zu lesen und zu schreiben. Im Gegensatz zu existierenden Thread-Debuggern wird damit zusätzlich die Möglichkeit geboten, Mutexe und Bedingungsvariablen zu erkennen. Weiterhin ist es möglich, Beziehungen zwischen den Objekten zu analysieren.

Auf der Grundlage des TDI ist es möglich, sogenannte High-Level-Debugger zu implementieren. Eine Erkennung von zyklischen Abhängigkeiten (Deadlock) oder das unerlaubte Schreiben von Daten (Data Races) ist mit Hilfe der vom TDI angebotenen Informationen realisierbar.

Der Zugriff auf die Daten der POSIX-Threads-Objekte durch das TDI erfolgt generisch. Um diesen generischen Zugriff zu gewährleisten, müssen Details der Implementierung von der jeweiligen Pthreads-Lösung gekapselt werden. Um die Kapselung zu gewährleisten, wurde in dieser Arbeit eine Erweiterung der POSIX-Threads definiert - das TED (*Thread Extension for Debugging*). Im TED sind Mengeniteratoren und Attributzugriffsfunktionen enthalten, die vom TDI-Kern benutzt werden können.

Bisherige Lösungen für das Thread-Debugging verlagern den gesamten Aufwand auf die Seite des Debuggers. Informationen über Threads werden gewonnen, indem die Thread-Strukturen binär aus dem Prozeßabblid der Applikation gelesen und auf der Seite des Debuggers gemäß der Typdefinition der jeweiligen Pthreads-Implementierung dekodiert werden. Im Hinblick auf die Menge der extrahierbaren Informationen ist diese Vorgehensweise eingeschränkt, da lediglich Thread-Informationen gewonnen werden können. Mutexe und Bedingungsvariablen werden vom Nutzer angelegt und verwaltet, sodaß sie für den Debugger nicht über definierte Symbolnamen zugreifbar sind.

Mit dem TDI wird der Ansatz der aktiven Debug-Unterstützung durch die Pthreads-Laufzeitumgebung verfolgt. TDI-Kern und TED bilden eine Art Informations-Server, der die in der hier definierten Anfragesprache TDI-QL spezifizierten TDI-Anfragen auswertet. Im Gegensatz zur konventionellen Variante des Datenaustausches zwischen Debugger und Applikation über die Betriebssystemschnittstelle `ptrace` kann die Applikation die gewünschten Daten über eine beliebige Form der Interprozeßkommunikation an den Debugger übermitteln, womit sich ein meßbarer Geschwindigkeitsvorteil ergibt, da die Zahl der für den Datenaustausch nötigen Systemrufe drastisch gesenkt wird.

Durch den hier verfolgten Ansatz der aktiven Debug-Unterstützung durch die Zielapplikation ist eine Kapselung der Pthreads-Daten und damit ein generischer Pthreads-Zugriff durch den TDI-Kern erst möglich, da die Funktionen zum Attributzugriff im Zielprozeß ausgeführt werden müssen.

Das Ausführen von Teilen der Applikation (TED) zum Zwecke des Debuggings birgt Gefahren in sich. Ein Scheduling muß während der TDI-Ausführung vermieden werden. In diesem Zusammenhang wurden eine eigenständige Speicherverwaltung und ein Mechanismus zur Deaktivierung des zeitscheibengesteuerten Schedulings bei User-Space-Implementierungen geschaffen.

Im Zuge dieser Arbeit entstand der TDI-Kern und es wurden die Pthreads-Implementierungen FSU-Pthreads, MIT-Pthreads und die Linuxthreads um die TED-Funktionalität ergänzt. Damit konnte die Realisierbarkeit des Ansatzes sowohl für User-Space- als auch für reine Kernel-Threads-Implementierungen der Pthreads demonstriert werden.

Um die Möglichkeiten aufzuzeigen, die sich für das Thread-Debugging durch die Benutzung des TDI ergeben, wurde der GNU-Debugger (Gdb) für die TDI-Nutzung angepaßt. Eine Anpassung weiterer Debugger ist aufgrund der einfachen TDI-Schnittstelle leicht möglich. Ein Thread-Debugger auf der Grundlage des TDI ist unabhängig von der Pthreads-Implementierung und damit portabel.

Die erfolgte Definition einheitlicher Schnittstellen zwischen Pthreads-Implementierung, TDI und Debugger ermöglicht eine unabhängige Implementierung der drei Komponenten. Eine Änderung von Implementierungsdetails unter Wahrung der Schnittstellen hat keinen Einfluß auf die Funktionsfähigkeit der jeweils anderen Komponenten. Eine Erweiterung der Debug-Unterstützung durch die Bereitstellung von TED-Modulen für weitere Pthreads-Implementierungen soll dadurch begünstigt und forciert werden.

2. Zusammenfassung

Threads sind leichtgewichtige Kontrollflußeinheiten innerhalb eines Adreßraumes. Die Benutzung von Threads ermöglicht es auf einfache Weise, leistungsfähige nebenläufige und parallele Systeme zu entwickeln. Die Standardisierung einer C-Schnittstelle zur nebenläufigen Programmierung durch das POSIX[1]-Gremium stellt die Basis für die Entwicklung portabler nebenläufiger beziehungsweise paralleler[2] Multithread-Anwendungen dar.

Der POSIX-Threads-Standard, kurz Pthreads, definiert ausschließlich die Programmierschnittstelle, das Debugging-Problem wird nicht adressiert. Somit besteht eine Diskrepanz zwischen einheitlichen Entwicklungs- und Testmöglichkeiten. Die bisherige Entwicklung proprietärer, uneinheitlicher Testwerkzeuge für POSIX-Threads-Anwendungen macht das deutlich. Die Uneinheitlichkeit resultiert aus der Tatsache, daß die existierenden Debugger zu sehr an die jeweilige Pthreads-Implementierung gebunden sind. Es fehlt ein einheitlicher Ansatz, der dem Debugger einen generischen Zugriff auf alle Pthreads-Implementierungen gestattet.

Diese Arbeit hat das Ziel, eine einheitliche Debug-Schnittstelle (*Thread Debug Interface*, kurz TDI) zu definieren und umzusetzen, mit der es möglich ist, Pthreads-Anwendungen auf verschiedenen Plattformen und bei Verwendung beliebiger standardkonformer Pthreads-Implementierungen zu debuggen. Diese Debug-Schnittstelle soll es ermöglichen, die im POSIX-Threads-Standard definierten Objekte und deren Attribute sowie vorhandene Objektbeziehungen für den Debugger und für dessen Nutzer sichtbar zu machen.

Zusätzlich zur Schaffung einer Debug-Schnittstelle bestand ein weiteres Ziel darin, einen existierenden Debugger und verschiedene POSIX-Threads-Implementierungen exemplarisch für die Schnittstelle anzupassen, um einerseits die Realisierung zu demonstrieren und andererseits eine Referenz für Änderungen an weiteren Debuggern und Pthread-Implementierungen zu schaffen. Im Ergebnis wurden der GNU-Debugger *Gdb* sowie die User-Space-Implementierungen FSU-Pthreads, MIT-Pthreads und die Kernel-Threads-Implementierung Linuxthreads für das TDI angepaßt.

[1] Das *Portable Operating System Interface for Computing Environments* (POSIX) ist eine Untermenge der IEEE-Standards.

[2] Auf Multiprozessoren können Threads parallel ausgeführt werden.

3. Gliederung

Kapitel 4, Einführung, gibt eine generelle Einführung in die Thematik. Es werden der POSIX-Threads-Standard, mögliche Formen seiner Implementierung und die Problematik des Debuggings nebenläufiger Programme betrachtet. Es wird auf die Vorteile für das Thread-Debugging eingegangen, die sich durch die Sichtbarkeit der POSIX-Threads-Objekte ergeben.

Die Anforderungen an das TDI sollen in Kapitel 5, Anforderungsdefinition, zusammengetragen werden. Die generelle Zielstellung, der künftige TDI-Einsatz, die Einsatzumgebung, der Schnittstellenzugang, funktionale und nichtfunktionale Anforderungen und das geforderte Verhalten im Fehlerfall sollen definiert werden [PB93]. Forderungen von Nutzern und Entwicklern ergeben sich sowohl auf der Seite der Pthreads als auch seitens des Debuggers.

Nachdem die Anforderungen an das *Thread Debug Interface* definiert wurden, wird die Architektur des TDI in Kapitel 6, Schnittstellenentwurf, festgelegt. Es werden alle Teilkomponenten und deren Zusammenwirken beschrieben.

Kapitel 7, Implementierung, beschreibt ausgewählte Details der Implementierung. Dabei werden Probleme der Einbettung des TDI-Kerns in die Pthreads-Laufzeitumgebung, die Realisierung der Anfragebearbeitung sowie die Änderungen einer Pthreads-Implementierung im Hinblick auf die TDI-Unterstützung betrachtet.

Die Erweiterung eines existierenden Debuggers auf der Grundlage des TDI wird in Kapitel 8, Thread-Debug-Erweiterung des Gdb, beschrieben.

In Kapitel 9, Schlußbetrachtungen, werden die Ergebnisse dieser Arbeit zusammengefaßt. Ideen und Vorschläge für zukünftig zu realisierende Änderungen und Erweiterungen des TDI werden zusammengetragen.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungen

POSIX – Portable Operating System Interface
API – Application Programming Interface
TDI – Thread Debug Interface
TED – Thread Extension for Debugging
TDI-QL – TDI Query Language
GDB – GNU Debugger
LWP – Light Weight Process
MT – Multithread

4. Einführung

Die Bereitstellung einer einheitlichen Debug-Schnittstelle für Implementierungen der POSIX-Threads ist mit einer Reihe von Themenkreisen verbunden, da sowohl die Schnittstelle als auch deren Benutzer (Debugger und Pthreads-Laufzeitumgebung) betrachtet werden müssen. Dieser Abschnitt soll einführenden Charakter haben und auf die in diesem Umfeld vorhandenen Fakten und Probleme hinweisen.

4.1. Motivation

Die Standardisierung einer C-Schnittstelle für das nebenläufige Programmieren brachte den POSIX-Standard 1003.1c hervor[IEE95]. Seither sind für die verschiedensten Plattformen standardkonforme Implementierungen entstanden [Mue93, PHR93, Ler96, Sunb]. Die Bereitstellung des Entwicklungswerkzeugs Debugger, das im Zusammenhang mit der Benutzung des POSIX-Threads-API erforderlich ist, vollzog sich jedoch nicht in einem für den Applikationsentwickler gewünschten Maße.

Der Umstand, daß es sich bei den POSIX-Threads um User-Level-Threads (siehe Abschnitt 4.1.2) handelt, ist die Hauptursache für die fehlende Debug-Unterstützung. Es gibt bisher kein Betriebssystem, daß eine Pthreads-Programmierung über die Systemschnittstelle erlaubt. Dem Betriebssystem fehlen daher jegliche Informationen über instantiierte POSIX-Threads. Die wenigen proprietären Lösungen für das Pthreads-Debugging sind zu stark an die einzelne POSIX-Threads-Implementierung angepaßt und häufig kaum auf andere Implementierungen übertragbar. Konsequenterweise sollte man den Nutzer der Pthreads in die Lage versetzen, seine Entwicklungswerkzeuge, zu denen ein Debugger als Testwerkzeug zählt, unabhängig von der konkreten Thread-Implementierung, sondern ausschließlich gebunden an den POSIX-Threads-Standard, nutzen zu können. Damit entsteht die Forderung nach einer standardisierten oder zumindest einheitlichen Thread-Debug-Schnittstelle für POSIX-Threads-Implementierungen.

Diese Arbeit hat die Bereitstellung einer einheitlichen Debug-Schnittstelle für Pthread-Implementierungen zum Ziel. Die Schnittstelle hat die Aufgabe, die einzelnen Kontrollflußeinheiten (Threads) sichtbar zu machen und darüber hinaus den gesamten durch das Pthreads-API definierten Blick auf die Implementierung zu ermöglichen.

Im folgenden soll auf den POSIX-Threads-Standard, seine Implementierungsformen und die sich ergebenden Probleme für das Thread-Debugging eingegangen werden.

4.1.1. POSIX-Threads-API

POSIX als Untermenge der IEEE-Standards definiert im Standard 1003.1c[1] eine Threadschnittstelle für die Programmiersprache C. Der Standard umfaßt die Namensdefinition für Objekte der nebenläufigen Programmierung und die Funktionen für das Threadmanagement, die Synchronisation und das Scheduling. Basisobjekte sind Thread, Mutex und Bedingungsvariable.

Threads sind das zentrale Ausdrucksmittel für die Nebenläufigkeit. Jeder Thread ist mit einer von ihm ausgeführten C-Funktion assoziiert, der ein threadspezifisches Argument übergeben werden kann. Zur Kontrolle und Instantiierung der Threads werden folgende Funktionen bereitgestellt:

- Erzeugung und Beendigung (`pthread_create` und `pthread_exit`),
- Identität und deren Vergleich (`pthread_self` und `pthread_equal`),
- Warten auf die Beendigung des Threads (`pthread_join`),
- Entkoppeln eines Threads vom Mutter-Thread (`pthread_detach`) und
- das Setzen und Lesen von Attributen (`pthread_attr_...`-Funktionen).

Mutex und Bedingungsvariable sind die vom Pthreads-Standard bereitgestellten Synchronisationsmittel. **Mutexe** können mit gemeinsam genutzten Ressourcen assoziiert werden, um deren Zugriff zu reglementieren. Der abstrakte Datentyp[2] Mutex bietet die Methoden

- Initialisieren und Vernichten (`pthread_mutex_init` und `..._destroy`),
- Anforderung und Freigabe (`pthread_mutex_lock`, `..._trylock` und `..._unlock`) sowie
- das Setzen von Attributen (`pthread_mutexattr_...`-Primitiven).

[1] Draft 5 erschien im April 1992 und wurde bis hin zu Draft 10 (Juni 1995) weiterentwickelt. 1996 erschien der endgültige Standard.

[2] Thread, Mutex und Bedingungsvariable werden im folgenden auch häufig als Objekte bezeichnet.

Mutexe werden benutzt, um kritische Abschnitte des Kontrollflusses zu schützen. Ist die Ausführung eines Threads an eine Bedingung geknüpft, deren Erfüllung zwingend für seine Ausführung ist, so kann er seinen Mutex an den Thread vergeben, der die Erfüllung dieser Bedingung beeinflußt. Bis zur Erfüllung des geforderten Zustandes ruht seine Ausführung. Das beschriebene Verhalten kann durch das Benutzen von **Bedingungsvariablen** (engl. condition variable) erreicht werden. Für diese dritte Klasse von Basisobjekten der POSIX-Threads wird die folgende Funtionalität angeboten:

- Initialisieren und Vernichten (`pthread_cond_init` und `..._destroy`),
- Warten auf Signalisierung (`pthread_cond_wait` und `..._timedwait`),
- Signalisieren (`pthread_cond_signal` und `..._broadcast`),
- Setzen von Attributen (`pthread_condattr_...`-Primitiven).

Ein weiterer wichtiger Teil der im Standard definierten Funktionen ist dem Scheduling gewidmet. Der **Scheduler** ist nicht direkt über das API zugreifbar. Auf das Scheduling kann nur über spezielle Thread-Attributfunktionen (`pthread_attr_set_...` und `pthread_attr_get_...`) und die Funktion `sched_yield` und implizit über blockierende Funktionsrufe von `pthread_mutex_lock` und `pthread_cond_wait` oder Ein- und Ausagbefunktionen[3] Einfluß genommen werden.

Im Hinblick auf ein Thread Debugging ist die Möglichkeit des Auslesens der Objektdaten ein wichtiges Kriterium. Der POSIX-Threads-Standard definiert die Semantik der Basisobjektfunktionen ohne Zuhilfenahme von statischen Objekteigenschaften. Einige Objektattribute werden implizit definiert. Zum Beispiel ist die von einem Thread ausgeführte C-Funktion ein ihn charakterisierendes Merkmal. Dieses Attribut kann aber nicht mit Hilfe der Thread-Attributfunktionen gelesen oder geschrieben werden. Andere Attribute, zum Beispiel die Priorität eines Threads, sind über Schreib- und Lesefunktionen zugreifbar (`pthread_attr_[get|set]prio`). Für ein Debugging müssen sowohl Attribute sichtbar gemacht werden, die über API-Funktionen gelesen und geschrieben werden können, als auch jene, die implizit durch den Standard definiert werden.

4.1.2. POSIX-Threads-Implementierungen

Bei den POSIX-Threads-Implementierungen unterscheidet man User-Space- und Kernel-Thread-Implementierungen [Tan95]. Beide Arten sollen durch die im Rahmen dieser Arbeit entwickelte Thread-Debug-Schnittstelle (TDI) unterstützt werden.

[3] Blockiert ein Thread im Laufe einer E/A-Funktion, so erhält der nächste lauffähige Thread CPU-Zeit.

4.1.2.1. POSIX-Threads im User-Space

Für Betriebssysteme, die keine Threads unterstützen, sind eine Reihe von User-Space-Implementierungen entstanden. In dieser Arbeit wurden exemplarisch die FSU-Pthreads [Mue93] und die MIT-Pthreads [PHR93] betrachtet.

Bei einer User-Space-Implementierung befindet sich die gesamte Laufzeitumgebung der Threads im User-Space. Die POSIX-Threads sind für das Betriebssystem nicht sichtbar und werden demzufolge nicht für das Scheduling berücksichtigt. Der Thread-Scheduler wird ebenfalls im User-Space ausgeführt, sodaß jede Applikation, die die Threadbibliothek benutzt, einen eigenen Scheduler besitzt. Die vollzogenen Thread-Kontextwechsel können ohne Zuhilfenahme des Betriebssystemkerns erfolgen und sind damit erheblich schneller als Prozeß-Kontextwechsel. Die vom Prozeß C in Abbildung 4.1 ausgeführte Applikation arbeitet auf der Basis einer User-Space-Implementierung der Pthreads. Alle User-Level-Threads sind Prozeß C zugeordnet und werden innerhalb des Prozesses gescheduled.

Das Thread-Scheduling auf Prozeßebene hat den Nachteil, daß sich alle Threads die dem Prozeß zugewiesene CPU-Zeit teilen müssen. Die Ausführungsbedingungen eines Threads hängen damit nicht nur von seinen Starteinstellungen (Priorität, Scheduler) ab, sondern müssen zusammen mit der Priorität des ihn ausführenden Prozesses und der Anzahl der im Prozeß instantiierten Threads betrachtet werden.

Eine Ausführungskontrolle der User-Space-Pthreads ist mit Betriebssystemmitteln nur eingeschränkt möglich. Diese erlauben es lediglich, den gesamten UNIX-Prozeß und damit alle Threads zu stoppen und auszuführen. Eine threadspezifische Ausführungskontrolle ist nicht möglich.

4.1.2.2. POSIX-Threads als Kernel-Threads-Aufsatz

Werden vom Betriebssystemkern Mittel bereitgestellt, nebenläufige Kontrollflüsse in einem gemeinsamen Adreßraum auszuführen, dann spricht man von Kernel-Threads.

Das Scheduling der Kernel-Threads erfolgt im Betriebssystemkern, das heißt alle Threads des Systems werden von einem Scheduler verwaltet. Damit haben gleichpriorisierte Threads im Gegensatz zu User-Space-Threads auch gleiche Ausführungsvoraussetzungen unabhängig von der Anzahl der Threads der jeweiligen Task[4].

Echte Kernelimplementierungen der POSIX-Threads sind in keinem UNIX-System vorhanden, wobei mit „echt" die Bereitstellung des Pthreads-API durch den Systemkern gemeint ist. Es gibt jedoch eine Reihe von Betriebssystemen (Mach, Solaris, Linux, VxWorks u.a.), die Threads als integralen Bestandteil unterstützen,

[4] Die Menge aller Threads, die auf einem Adreßraum operieren, wird als Task bezeichnet.

sodaß man POSIX-Threads realisieren kann, indem man sie von Kernel-Threads ausführen läßt.

Zum Studium dieser Technik diente die POSIX-Threads-Implementierung unter Linux [Ler96]. Bei den Linuxthreads wird jeder Thread intern mit einem `clone()`-Ruf erzeugt, sodaß jeder POSIX-Thread mit einem Kernel-Thread (UNIX-Prozeß) assoziiert ist. Der in Abbildung 4.1 dargestellte Prozeß A führt eine auf Linuxthreads basierende Applikation aus. Da jeder User-Level-Thread einem Kernel-Thread fest zugeordnet ist, ist kein Scheduling im User-Space nötig. Lediglich die Kernel-Threads werden vom System-Scheduler verwaltet.

Auf Multiprozessoren mit gemeinsamem Speicher können Kernel-Threads parallel ausgeführt werden. Die parallele Ausführung von POSIX-Threads ist durch die Bindung an Kernel-Threads möglich, da die Kernel-Threads im Systemkern sichtbar sind und ihnen vom Systemscheduler eine CPU zugeordnet werden kann. Pthreads innerhalb eines Prozesses (User-Space) können nicht parallel auf Multiprozessoren ausgeführt werden, wenn der Prozeß die vom Scheduler betrachtete kleinste Einheit ist.

4.1.2.3. Mischformen

Die Zuordnung von User-Level-Threads[5] zu Kernel-Threads läßt sich wie folgt verallgemeinern. Sei ein Thread-Cluster (auch Task genannt) C gegeben, der alle Threads eines gemeinsamen Adreßraumes enthält. Auf oberster Ebene wird eine Menge U von User-Level-Threads ausgeführt. Diese User-Level-Threads werden einer Menge K von Kernel-Threads zur Ausführung zugeordnet. Die Abbildung 4.1 zeigt mögliche Zuordnungen zwischen User-Level- und Kernel-Threads.

Ein Thread-Cluster C kann als ein Tupel $C = (U, K)$ aufgefaßt werden. POSIX-Threads-Implementierungen lassen sich wie folgt klassifizieren. Bei reinen User-Level-Implementierungen ist $|K|$ immer 1. Bei reinen Kernel-Thread-Implementierungen ist $|K| = |U|$, wobei jedes Element aus U von genau einem Kernel-Thread aus K ausgeführt wird. Bei dieser Lösung müssen nur die Kernel-Threads gescheduled werden. Alle anderen Implementierungen, bei denen Cluster zulässig sind, die nicht den genannten Kriterien entsprechen, handelt es sich um Mischformen. Hier erfolgt ein Scheduling sowohl im User-Space als auch im Betriebssystemkern.

Ein Beispiel für eine Mischform sind die Solaris-Pthreads [Sunb]. Hier wird jeder Prozeß als ein Thread-Pool betrachtet, dem eine gewisse Anzahl von LWP's (virtuelle Prozessoren) zugeordnet wird, die kleiner als die Zahl der POSIX-Threads sein kann. Es existiert damit keine feste Bindung zwischen Kernel- und POSIX-Threads. Ein Scheduler im User-Space teilt den POSIX-Threads die LWP's zu. Die Ausführung der LWP's wird vom Systemscheduler kontrolliert.

[5] Bei den betrachteten Pthreads-Implementierungen sind die POSIX-Threads immer User-Level-Threads, da sie im Nutzer-Adreßraum erzeugt und verwaltet werden.

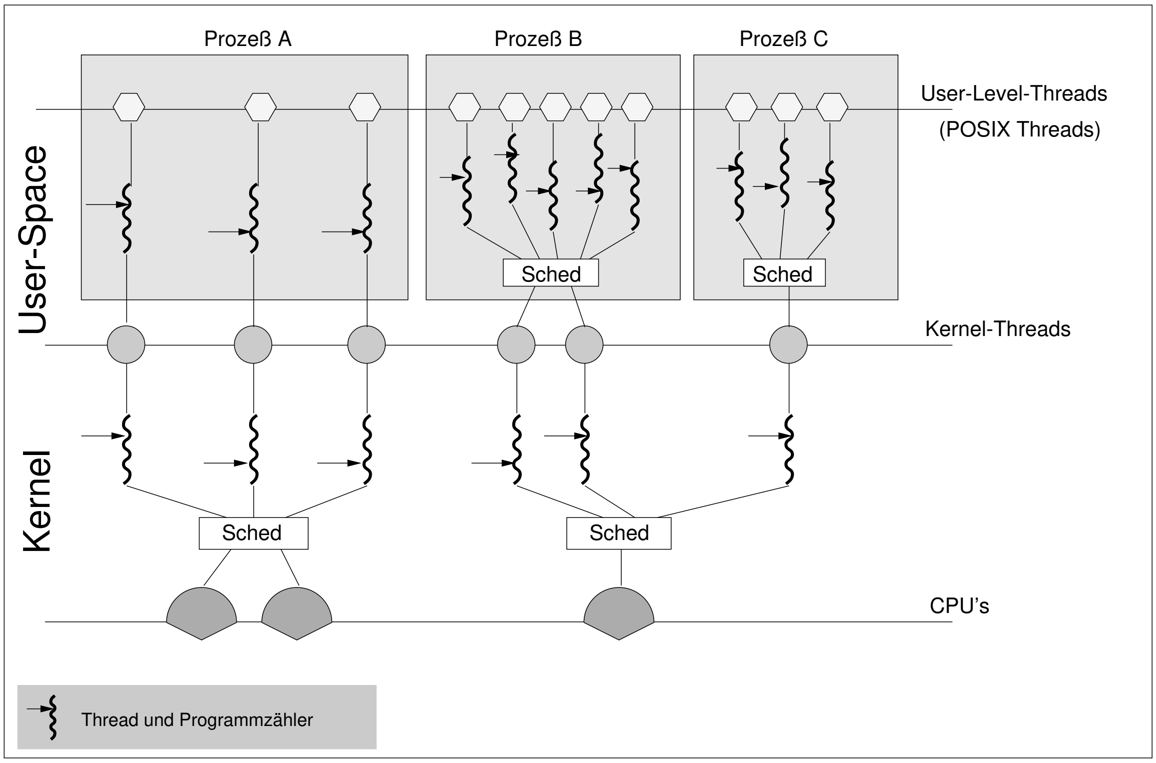

Abbildung 4.1.: Implementierungsformen für POSIX-Threads

4.1.2.4. Gemeinsamkeiten im Hinblick auf eine einheitliche Schnittstelle

Trotz der Unterschiede hinsichtlich der möglichen Cluster-Formen haben alle Implementierungsvarianten Gemeinsamkeiten, die Einfluß auf den Entwurf der Thread-Debug-Schnittstelle TDI haben.

POSIX-Threads sind generell User-Level-Threads, das heißt sie werden im User-Space erzeugt und beendet.

Alle Pthreads-Lösungen sind in Form von Funktionsbibliotheken verfügbar und nicht Bestandteil des Betriebssystemkerns. In Bezug auf die Unterstützung des TDI sind demzufolge Änderungen der Pthreads-Implementierung, ob von kommerziellen Anbietern oder unabhängigen Entwicklergemeinschaften, ohne fundamentale Änderungen am System möglich.

Alle im POSIX-API definierten Funktionen werden im User-Space ausgeführt. Die Daten für Mutexe und Bedingungsvariablen werden im User-Space erzeugt und ausschließlich vom Nutzer in seiner Applikation verwaltet. Lediglich die Thread-Daten werden aus Gründen des Schedulings und der Stackallokation von der Implementierung zusätzlich verwaltet.

Die Daten der Pthreads-Objekte liegen im Adreßraum des Clusters vor. Damit ergibt sich hinsichtlich des Zugriffs auf die Pthreads-Objekte eine Vereinfachungsmöglichkeit. Für den Zugriff auf die Pthreads-Daten der Applikation ist es ausreichend, einen Kernel-Thread des Thread-Clusters zu selektieren, da allen Threads der gesamte Adreßraum des Clusters zugeordnet ist. Mit Hilfe eines herkömmlichen prozeßgebundenen Debuggers ist es möglich, das Abbild dieses Kernel-Threads zu manipulieren und für die TDI-Ausführung zu nutzen.

4.1.3. Thread Debugging

Ein wesentlicher Teil dieser Arbeit ist der Schaffung eines Thread-Debuggers auf der Basis des im Mittelpunkt stehenden *Thread Debug Interface* (TDI) gewidmet. Deshalb soll hier etwas näher auf die Probleme des Debuggings nebenläufiger und paralleler Applikationen eingegangen werden.

Ein sequentielles Programm hat nur einen Kontrollfluß, sodaß alle jemals ausgeführten Teile des Programms nacheinander abgearbeitet werden. Bei einem nebenläufigen Programm gibt es mehrere quasi parallel existierende Kontrollflüsse (Threads). Für das Debugging von Multithread-Applikationen ergeben sich bedingt durch die Kontrollflußparallelität die im folgenden beschriebenen Probleme.

Kontrollflußselektion

Zum einen gibt es ein Problem, wenn zwei Threads dasselbe Stück Programmcode ausführen. Setzt man in diesem Programmstück einen Breakpoint[6], so ist nicht determiniert, welcher Thread dort zuerst stoppt. Eine Forderung besteht also darin, threadspezifische Breakpoints setzen zu können. So ist es dann möglich, den Kontrollfluß eines speziellen Threads zu selektieren.

Kontrollflußbindung

Ein weiteres Problem im Zusammenhang mit den angesprochenen parallelen Kontrollflüssen besteht darin, einen Kontrollfluß zu verfolgen. Seien also wieder zwei Threads gegeben, denen dasselbe Codestück zugeordnet ist. Einer der beiden Threads hat einen Breakpoint erreicht. Benutzt man einen herkömmlichen Debugger, um das Programm bis zur nächsten Quellzeile auszuführen, so kann nicht gewährleistet werden, daß die Applikation wieder im selben Thread stoppt. Es werden deswegen Debugger-Kommandos zur Programmausführung benötigt, die den Kontrollfluß eines einzelnen Threads verfolgen.

[6]Ein Breakpoint ist eine durch den Debugger eingefügte Instruktion im Programmtext, bei deren Ausführung das Programm stoppt.

Verfolgung des Scheduling

Sollte es möglich sein, threadspezifische Breakpoints zu setzen und den Kontrollfluß eines Threads zu verfolgen, so wäre es umgekehrt wieder interessant, das Scheduling zu beobachten. So könnte der Nutzer feststellen, welche Threads neben der Ausführung des selektierten Threads CPU-Zeit erhielten.

In Abbildung 4.2 ist das räumliche und zeitliche Verhalten zweier Threads in einem gemeinsam benutzten Programmstück skizziert. Der Nutzer hat einen Breakpoint bei der Codeaddresse I_1 gesetzt. Nach dem Start des Programmes stoppt Thread B bei I_1 zum Zeitpunkt t_1. Der herkömmliche Debugger kann nicht zwischen Threads unterscheiden, und demzufolge können dem Nutzer auch keine Informationen über den aktiven Thread erteilt werden. Damit ergibt sich als grundlegende Forderung für das Thread-Debugging die Sichtbarkeit der im POSIX-Standard eingeführten Objekte Thread, Mutex und Bedingungsvariable. Angenommen, der Nutzer besitzt Kenntnis über den aktuellen Thread B. Mit der Absicht, den Kontrollfluß von Thread B weiterzuverfolgen, führt er das Kommando `next`[7] aus. Da Thread B im Rahmen des Scheduling suspendiert wird, stoppt die Applikation erneut bei Instruktion I_1 zum Zeitpunkt t_2. Der aktive Thread ist jedoch Thread A. Die vom Nutzer gewünschte Verfolgung des Kontrollflusses von Thread B konnte damit nicht realisiert werden. Aufgabe eines Thread-Debuggers ist es, die Befehle, die sonst zur Verfolgung des Kontrollflusses eines Prozesses dienen, an einen Thread zu binden. Wünscht der Nutzer keine Threadbindung von Debugging-Kommandos, so könnte eine Option darin bestehen, die Kontextwechsel der Threads, wie in Abbildung 4.2 als $tc_{1,2,3}$ dargestellt, anzuzeigen. Damit hat der Nutzer zwar nicht die Möglichkeit, einen Thread zu selektieren und dessen Kontrollfluß zu verfolgen, der Schedule wird damit jedoch sichtbar. Die Sichtbarkeit der Kontextwechsel und die Kontrollflußbindung sind orthogonale Debugging-Methoden und können demzufolge nebeneinander eingesetzt werden.

In Kapitel 8 wird die Anpassung eines herkömmlichen Debugger für das Thread-Debugging auf Grundlage des hier entwickelten TDI beschrieben.

4.2. Verwandte Forschungsarbeiten

McDowell und Helmbold [MH89] geben eine generelle Einführung in die Thematik des Debuggings nebenläufiger und verteilter Programme. Es werden breakpoint- und ereignisorientierte Debugger sowie Werkzeuge zur statischen Analyse betrachtet. Ein Schwerpunkt der Arbeit lag in der Untersuchung der jeweils durch den Ansatz bedingten Möglichkeit des „probe effects“ [Gai86] und seiner Adressierung in existierenden Debuggern der jeweiligen Klasse. Als Zielarchitekturen wurden sowohl Shared-Memory- als auch Message-Passing-Systeme betrachtet. Bei der

[7] Das Kommando next bewirkt die Ausführung des Programmes bis zur nächsten Quellzeile.

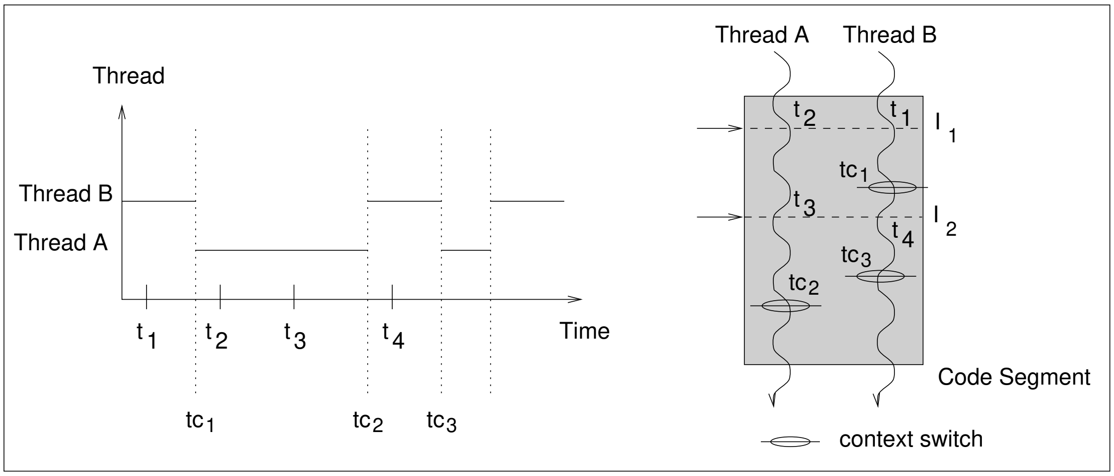

Abbildung 4.2.: Kontrollflüsse zweier Threads zeitlich und räumlich

Verwendung herkömmlicher Debugger (Breakpoint-Debugger, sequentieller Debugger) für das Problem der Nebenläufigkeit wurde nur der Fall betrachtet, daß die Threads oder Prozesse vom Betriebssystem sichtbar sind. Ein Debugging dieser Form der Nebenläufigkeit ist mit Hilfe von Gruppen sequentieller Debugger möglich, die zentral von einem Prozeß koordiniert werden. Auf das Problem des Thread-Debuggings für User-Level-Threads wurde nicht eingegangen.

4.2.1. Mach Debugger

Einen Thread-Debugger für die UNIX-Emulation unter Mach stellen Caswell und Black [CB90] vor. Der UNIX-Systemruf `ptrace` wurde wurde im Hinblick auf die Erkennung und Kontrolle einzelner Threads modifiziert. In der ursprünglichen `ptrace`-Version wurde vorausgesetzt, daß eine Anwendung nur auf UNIX-Prozesse erfolgt, die gewöhnlich durch eine Task mit genau einem Thread realisiert werden.

Die vom Mach-Kern bereitgestellten Befehle zur Task- und Threadkontrolle `task_resume`, `thread_resume` und `task_suspend`, `thread_suspend`, fanden bei der `ptrace`-Modifikation Verwendung, um einzelne Threads oder die ganze Applikation kontrollieren zu können. Für den Zugriff auf den Speicher eines bestimmten Threads wurden die Funktionen `vm_read` und `vm_write` genutzt, sodaß verglichen zur alten Lösung eine Verminderung von Kernelzugriffen erreicht werden konnte[8].

Der im Ergebnis entstandene Multithread-GDB beinhaltet eine Reihe neuer Befehle für das Thread-Debugging. Die Selektion eines bestimmten Threads sowie threadspezifische Breakpoints sind möglich.

[8] Mit `ptrace` können lediglich einzelne Speicherworte gelesen und geschrieben werden. Die vm-Funktionen erlauben den Zugriff auf größere Datenmengen bei geringerer Zahl an Funktionsrufen.

Der entstandene Thread-Debugger ist nicht portabel, da seine Implementierung auf einem geänderten nicht standardgemäßen `ptrace` basiert. Die Lösung ist weiterhin eingeschränkt auf die Benutzung von Kernel-Threads. User-Level-Threads wie die Pthreads, die von einem Kernel-Thread ausgeführt und gescheduled werden, werden in dieser Arbeit nicht adressiert.

4.2.2. MpD

Ponamgi, Hseush und Kaiser [PHK91] stellen eine Arbeit vor, die sich mit dem Erkennen von Event-Mustern durch den von Caswell und Black erweiterten Debugger beschäftigt. Hier werden sogenannte DPE's (Data Path Expressions) und PA's (Predecessor Automata) zum Erkennen von fehlerhaften Programmzuständen, wie zum Beispiel Deadlocks, Livelocks oder die Verletzung des gegenseitigen Ausschlusses, genutzt. Damit wird der GDB als ein typischer Vertreter der Breakpoint-Debugger um die Funktionalität eines Event-basierten Debuggers erweitert. Die DPE's müssen vom Nutzer definiert werden. Dafür wurde ein spezieller Interpreter für die DPE-Sprache in den GDB integriert. Durch die Programmierbarkeit konnte eine höhere Flexibilität des Debuggers erreicht werden. Allerdings muß der Programmierer im Vorfeld des Debugging genaue Kenntnis über mögliche auftretende Fehler haben, um DPE's spezifizieren zu können.

4.2.3. SmartGDB

Der *SmartGDB*[Hal96] ist ein Thread-Debugger auf der Grundlage des GDB. Die *SmartGDB*-Lösung hat den Anspruch, portabel, leicht erweiterbar und nutzerorientiert zu sein. Es werden verschiedene Thread-Pakete unterstützt, wobei aber keine Beschränkung auf Kernel- oder User-Level-Threads erfolgt. Aufgrund der Universalität des *SmartGDB* stehen keine speziellen Mittel für das Debugging von POSIX-Threads zur Verfügung.

Um eine höchstmögliche Anpassung an die Gewohnheiten des Nutzers zu erreichen, wurden die Ausdrucksmöglichkeiten der Kommandosprache des GDB um die Mittel der Skriptsprache *Tcl* und der Datenanfragesprache *DUEL* erweitert. Ein *Tcl*-Interpreter wurde so integriert, daß die Befehle beider Werkzeuge, Gdb und *Tcl*-Interpreter, in einer Kommandooberfläche benutzbar sind. Mit dieser Lösung werden den Anwendungen die Fähigkeiten der jeweils anderen übertragen. Nachteilig ist jedoch, daß Namenskonflikte der Befehle durch Umbenennung der Gdb-Kommandos gelöst wurden. DUEL[GH93] ist eine spezielle Anfrageerweiterung für das Debugging, die eine komfortable Untersuchung des derzeitigen Programmzustandes ermöglicht.

Die vom *SmartGDB* angebotene Thread-Debug-Unterstützung umfaßt die folgenden Basisbefehle:

- Identifizieren aller Threads einer Applikation,
- Anzeige des Thread-Zustandes,
- threadspezifische Breakpoints,
- Anzeige des Ausführungsstacks eines Threads und
- das Bewirken eines Kontextwechesels (nur User-Level).

Für die Umsetzung der Befehle wurden sowohl am GDB als auch in den unterstützten Thread-Implementierungen Änderungen vorgenommen. Insbesondere müssen für die Umsetzung des Kontextwechsels Funktionen in den Thread-Bibliotheken bereitgestellt werden, die die internen Warteschlangen manipulieren.

Die Unterstützung des *SmartGDB* für User-Level-Implementierungen basiert auf dem Dekodieren der internen Datenstrukturen der Thread-Implementierung im GDB. Dafür wird vorausgesetzt, daß die entsprechende Thread-Bibliothek mit Symbolinformationen vorliegt. Ein wesentlicher Nachteil dieser Lösung liegt in der Sensitivität gegenüber Namens- oder Typänderungen interner Variablen. Diesbezügliche Änderungen erfordern eine Reimplementierung der Dekodierungsroutinen des *SmartGDB*.

4.2.4. Debugging von User-Level-Threads unter Solaris

Unter Solaris werden zwei Arten von User-Level-Threads bereitgestellt, Solaris-Threads und POSIX-Threads. Für beide Implementierungen wird eine Debug-Unterstützung angeboten. Über das `/proc`-Dateisystem ist es möglich, die Daten des Zielprozesses (UNIX-Prozeß) auszulesen. Weiterhin ist es mit Hilfe des Systemrufes `ioctl` möglich, die Ausführung des Zielprozesses oder einzelner LWP's zu kontrollieren. Die Operationen können auf den Deskriptoren der mit den LWP's oder Prozessen assoziierten `/proc`-Dateien ausgeführt werden. Der Systemruf `ptrace` ist unter Solaris im Gegensatz zur vorgestellten Lösung unter Mach weiterhin auf Prozesse beschränkt.

Neben der Ausführungskontrolle für die Kernel-Threads wird die Bibliothek `libthread_db.so.1` für die Gewinnung von Thread-Informationen bereitgestellt. Die Funktionen der Bibliothek dekodieren ähnlich wie beim *SmartGDB* das Speicherabbild des Zielprozesses. Es werden Funktionen zur Iteration über die Thread-Menge sowie Attributlesefunktionen bereitgestellt. Mutexe und Bedingungsvariablen wurden nicht berücksichtigt.

In einer Erweiterung des GDB [Sta89] wurden die bereitgestellten Mittel benutzt, um ein Thread-Debugging für Solaris-Threads und Solaris-Pthreads zu ermöglichen. Die Debug-Erweiterung bietet für die POSIX-Threads die gleiche Funktionalität, wie für die proprietären Solaris-Threads. Die Objekte des POSIX-Thread-

Standards und deren Beziehungen werden nicht sichtbar gemacht. Weiterhin ist die Lösung eingeschränkt auf die Solaris-Implementierung.

4.2.5. KDB - Ein Thread-Debugger für μC++

Der KDB[BKS96] ist ein Mutithread-Debugger, der Kernel- und User-Level-Threads unterstützt. Er wurde für die C++-Erweiterung μC++ entwickelt, die Sprachkonstrukte für die Nebenläufigkeit bereitstellt. Die μC++–Klassenbibliothek verfügt über Mittel zur Erzeugung von User-Level-Threads und Kernel-Threads, wobei mehrere User-Level-Threads einem Kernel-Thread zugeordnet werden können. Der KDB ist in der Lage, beide Thread-Formen in beliebiger Zuordnung zu debuggen.

Die KDB-Architektur unterscheidet sich grundlegend von der anderer Thread-Debugger. Es wird unterschieden zwischen lokalen Debuggern in der Zielapplikation und dem Main-Debugger, der mit dem Nutzer in Interaktion tritt. Der Main-Debugger ist seinerseits in Threads unterteilt, die jeweils einem lokalen Debugger in der Zielapplikation zugeordnet sind. Die inhärente Synchronität der Betriebssystemmittel `ptrace` und `/proc` (siehe Abschnitt 6.4.1) gilt damit nicht mehr für das Debugging des gesamten Zielprozesses, sondern lediglich für einen Kernel-Thread der Zielapplikation. Somit kann erreicht werden, daß die Ausführung von Kernel-Threads separat kontrolliert werden kann. Die separate Ausführung von User-Level-Threads ist betriebssystembedingt nicht möglich, da nur Mittel für die Ausführungskontrolle von Kernel-Threads zur Verfügung stehen.

Neben der neuartigen Architektur des KDB verfügt er über ein interessantes Detail im Zusammenhang mit dem Breakpoint-Debugging. Bei nebenläufigen Applikationen, deren Threads sich das Code-Segment teilen, ist es möglich, daß die mit dem Breakpoint verbundene Trap-Instruktion von allen Threads ausgeführt wird. Bei der herkömmlichen Methode wird der Debugger vom Betriebssystem benachrichtigt, wenn die Applikation einen Breakpoint erreicht hat. Handelt es sich um einen threadspezifischen Breakpoint, so muß der Debugger den spezifizierten Thread mit dem auslösenden Thread vergleichen und gegebenenfalls die Applikation ohne Interaktion mit dem Nutzer fortführen. Je höher die Zahl der Threads ist, die den Code-Abschnitt mit dem enthaltenen Breakpoint ausführen, um so höher ist der Laufzeitverbrauch für das Debugging. Der KDB benutzt die Methode der „fast breakpoints" [Kes90], um den Laufzeitverbrauch drastisch zu senken. Statt des Stopps der Applikation bei jedem Breakpoint wird eine Routine des lokalen Debugger gerufen, die den Thread-Vergleich durchführt. Die Auswertung eines Breakpoints, bei dem kein Applikationsstopp notwendig ist, ist im Vergleich zur herkömmlichen Lösung rund 1000 mal schneller.

4.2.6. Zusammenfassung

Keine der vorgestellten Lösungen nutzt die Möglichkeiten der POSIX-Threads-Standardisierung. Informationen über Threads und die Synchronisationsmittel Mutex und Bedingungsvariable sowie deren Beziehung untereinander könnten einheitlich und unabhängig von der jeweiligen Pthreads-Implementierung verwendet werden. Die Funktionalität von *SmartGDB* und *MpD* zur Erkennung von Deadlocks und Data Races könnte auf der Grundlage dieser Informationen einheitlich und generisch realisiert werden. Die grundlegenden Thread-Debug-Funktionen für die Threaderkennung, die Kontrollflußselektion und -bindung sowie für die Beeinflussung und Anzeige des Scheduling können ebenfalls mit Hilfe der POSIX-Threads-Informationen und damit unabhängig von der jeweiligen Implementierung erbracht werden.

Alle vorgestellten Thread-Debugger sind auf der Grundlage des GDB entstanden. Er bietet Low-Level-Routinen zum Debugging von Programmen für verschiedene Quellsprachen und ist auf den meisten UNIX-Plattformen lauffähig. Auch in dieser Arbeit wurde der GDB verwendet, um die Einbettung der entwickelten Thread-Debug-Schnittstelle zu demonstrieren.

5. Anforderungsdefinition

Die Anforderungsdefinition oder Systemspezifikation [PB93] soll alle Anforderungen beschreiben, die sich im Zusammenhang mit der Thread-Debug-Schnittstelle ergeben. Im Mittelpunkt stehen die Fragestellungen, was das TDI leisten beziehungsweise nicht leisten soll und welche Prämissen für seinen Einsatz bestehen.

5.1. Zielsetzung

Ziel dieser Arbeit ist es, eine Schnittstelle bereitzustellen, mit der es möglich ist, den Zustand einer Pthreads-Laufzeitumgebung zu erfassen und zu ändern. Die Menge der instantiierten POSIX-Threads-Objekte, deren Attributwerte sowie die Objektbeziehungen sind hier die Ausdrucksmittel, um den Zustand einer Multithread-Applikation zu beschreiben. Darin unterscheidet sich dieser Ansatz von bisherigen Thread-Debug-Lösungen (*SmartGDB* [Hal96], KDB [BKS96]), die keine Zustandserfassung und -änderung auf Pthreads-Abstraktionsniveau zulassen.

Mit dem TDI wird nicht die Zielstellung verfolgt, eine Basis-Debug-Unterstützung für Multithread-Anwendungen zu bieten. Das TDI soll vielmehr eine Erweiterung der Pthreads-Schnittstelle darstellen, die von herkömmlichen Debuggern genutzt werden kann, um ein einheitliches Thread-Debugging für POSIX-Threads zu ermöglichen.

5.2. TDI-Einsatz und TDI-Umgebung

Da POSIX ein UNIX-Standard ist, wird das TDI auf UNIX-Plattformen zum Einsatz kommen. Eine Debug-Unterstützung soll sowohl für Kernel- als auch für User-Space-Implementierungen der POSIX-Threads realisiert werden. Dabei wird vorausgesetzt, daß die Kernel-Threads oder Prozesse, die die POSIX User-Level-Threads ausführen, mit Hilfe herkömmlicher Debugger kontrollierbar sind. Das TDI hat lediglich die Aufgabe, die POSIX User-Level-Threads sichtbar und zugreifbar zu machen.

5.3. Schnittstellenzugang

Eine Forderung an die Schnittstelle ist ein von der Quellsprache des Debuggers möglichst unabhängiger Schnittstellenzugang. Die Praxis zeigt, daß der Debugger nicht unbedingt in der Sprache entwickelt ist, für die er eingesezt werden kann. Einige Debugger, beispielsweise der *GDB*, beherrschen mehrere Zielsprachen. Eine naheliegende Möglichkeit wäre es gewesen, eine Menge von Funktionen zur Verfügung zu stellen, die vom Debugger für den Zugriff auf die Schnittstelle verwendet werden können. Für die Vielzahl der verwendeten Quellsprachen hätte das jedoch einen großen Aufwand dargestellt. Betrachtet man die Funktionsaufrufe und das Auswerten der Ergebnisse als Datenaustausch zwischen Debugger und Schnittstelle, so ist es naheliegend, ein festgelegtes Protokoll zum Datenaustausch zu verwenden. Die Zahl der dafür notwendigen Primitiven ist fest. Das heißt inbesondere, daß sich auch bei einer Erhöhung des Funktionsumfanges des Schnittstellenkerns die Anzahl der Primitiven nicht ändert. Auf die Menge der von der Schnittstelle angebotenen Funktionen kann über entsprechende Nachrichten zugegriffen werden. Schnittstellenerweiterungen haben dann keinen Einfluß mehr auf den Schnittstellenzugang, lediglich die Menge der möglichen Nachrichten erhöht sich. In Abbildung 5.1 ist eine Anfrage des Debuggers an das TDI dargestellt. Für den Schnittstellenzugang sind auf der Seite des Debuggers die Primitiven

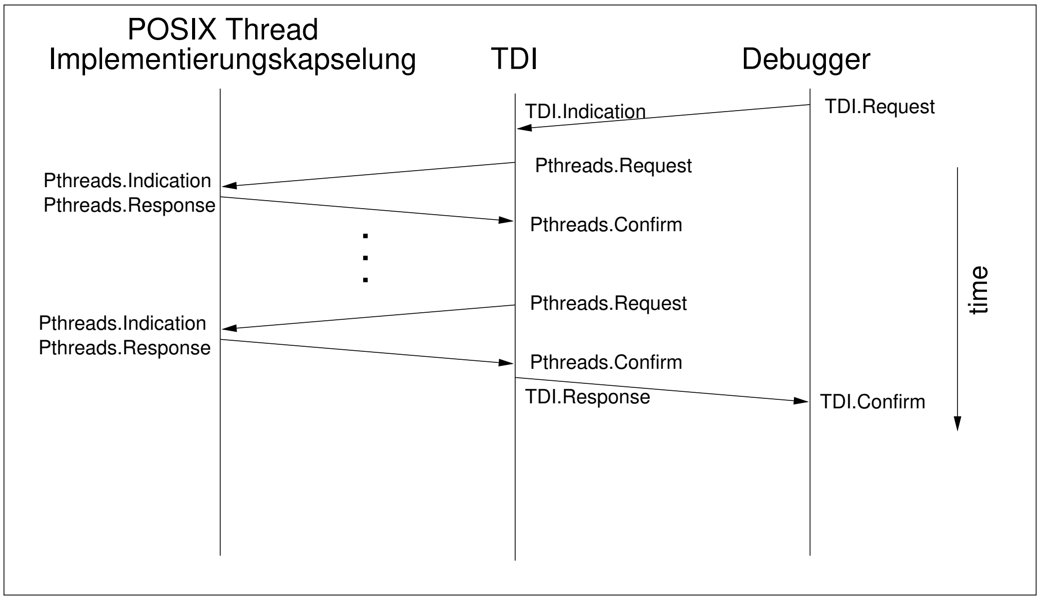

Abbildung 5.1.: Interaktion von Debugger und TDI

`TDI.request` und `TDI.confirm` notwendig. Mit `TDI.request` wird der Schnittstelle eine Nachricht übermittelt, die ein Befehl oder eine Anfrage sein kann. Je nach Nachrichtentyp wird mit `TDI.confirm` eine erfolgreiche Befehlsausführung oder das Ergebnis der Anfrage entgegengenommen. Detailfragen der Kommunikation zwischen Debugger und Schnittstelle werden in Abschnitt 6.4 diskutiert.

Um eine hohe Flexibilität der Schnittstelle zu gewährleisten, sollten die Anfragenachrichten in einer Anfragesprache formuliert werden können. Eine explizite Definition von möglichen Nachrichten wird durch die Festlegung der Grammatik der Anfragesprache ersetzt. In den Abschnitten 6.3.2 und 6.3.3 wird auf die Anfragemöglichkeiten und deren Formulierung eingegangen.

5.4. Funktionale Anforderungen

Das TDI hat die Funktion, Anfragen des Debuggers zu bearbeiten. Bezüglich dieser Aufgabe kann man das TDI in einen generischen und einen Pthreads-spezifischen Teil untergliedern. Der generische Teil wird im folgenden TDI-Kern genannt. Die Abbildung 5.2 zeigt die Unterteilung der Schnittstelle und ihre Einbettung.

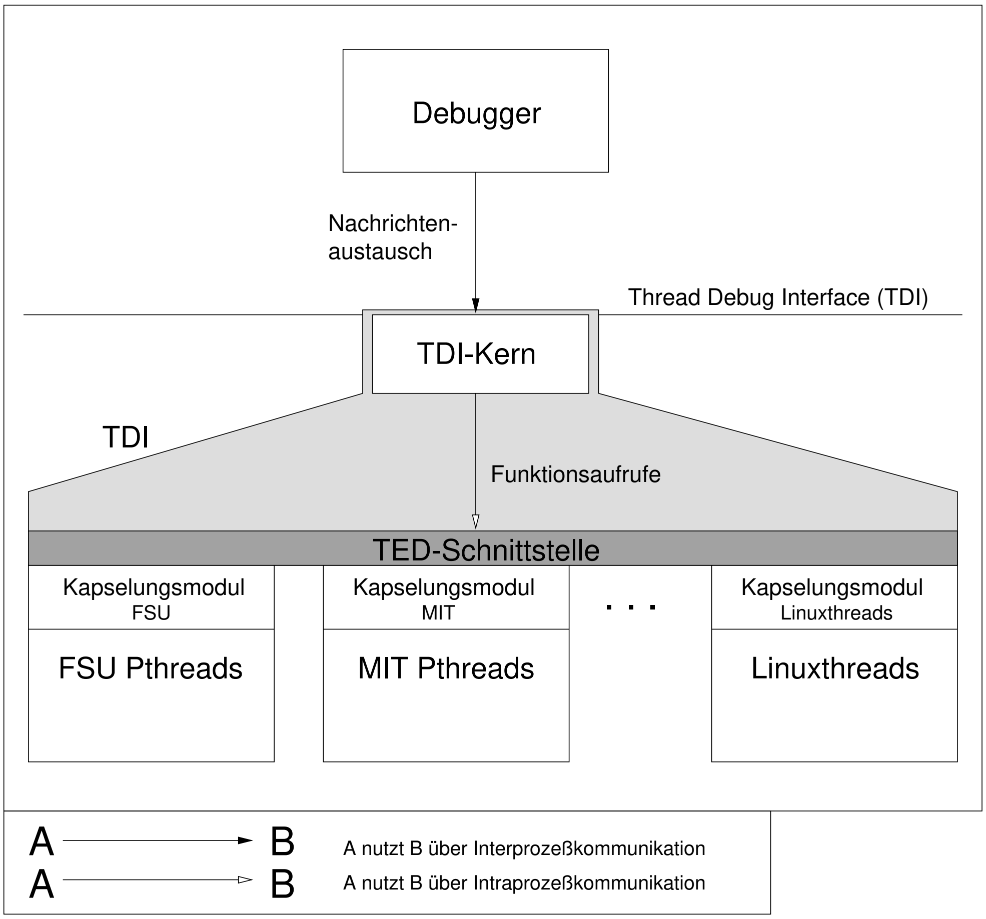

Abbildung 5.2.: Einbettung der Thread-Debug-Schnittstelle

Vom TDI-Kern werden nur Aufgaben erfüllt, die von der Pthreads-Implementierung unabhängig sind. Dazu gehören die Kommunikation und die Anfragebearbeitung. Die für die Anfrageauswertung notwendigen Zugriffsfunktionen auf die Daten der Pthreads-Laufzeitumgebung müssen in der jeweiligen POSIX-Threads-Implementierung durch eine Kapselungsschicht bereitgestellt werden. Die Kapselungsschicht stellt eine Erweiterung des POSIX-Threads-API dar und wird im folgenden TED (**T**hread-**E**rweiterung für das **D**ebugging) genannt.

Die Verteilung der Schnittstellenfunktionalität auf den TDI-Kern und die Pthread-Implementierungen hat den Vorteil, daß der generische TDI-Kern nur einmal implementiert werden muß und für alle Pthreads-Implementierungen benutzt werden kann. Die Kapselungsmodule müssen für jede einzelne Pthreads-Implementierung entwickelt werden. Die Entwicklung der TDI-Schichten kann unabhängig voneinander geschehen, lediglich die in Abschnitt 6.6 beschriebene Schnittstelle zwischen TDI und TED muß eingehalten werden.

5.5. Nichtfunktionale Anforderungen

Die nichtfunktionalen Forderungen an die Schnittstelle sind vielfältig, da sowohl die Entwickler der Pthreads-Implementierung und des Debuggers als auch die Nutzer Anforderungen stellen.

Portabilität

Das TDI ist für den Einsatz auf einem POSIX-konformen UNIX-System vorgesehen. Bei der Implementierung des TDI sollte daher nur die Standard-Betriebssystemschnittstelle (POSIX) benutzt werden, um die Ausführbarkeit auf allen Plattformen zu gewährleisten.

Trotz der Standardisierung gibt es Parameter für die TDI-Ausführung, die plattform- und rechnerabhängig sind. Hierzu zählen Ressourcenbeschränkungen wie beispielsweise die maximale Größe eines Shared-Memory-Segmentes für die Interprozeßkommunikation (IPC). Stellen diese Konstanten die Ausführbarkeit des TDI auf einer Plattform in Frage, so sollten sie konfigurierbar sein.

Alle systemabhängigen Berechnungen müssen von der Pthreads-Implementierung oder vom Debugger ausgeführt werden. Beispielsweise kann das TDI den Wert des aktuellen *pc* (Programmzähler) eines Threads an den Debugger weiterleiten. Das Lesen dieses Registers aus dem entsprechenden Thread-Kontext muß in der Pthreads-Implementierung geschehen.

Erweiterbarkeit

Die Funktionalität der Thread-Debug-Schnittstelle muß erweiterbar sein. Im Rahmen dieser Arbeit wurde die Basisfunktionalität implementiert. Sollten in Zukunft weitere zustandscharakterisierende Daten der Pthreads-Laufzeitumgebung erforderlich sein, so muß der Schnittstellenkern mit geringem Aufwand änderbar sein.

Eine Änderung des nachrichtenorientierten Schnittstellenzuganges (`TDI.request` und `TDI.confirm`) darf nicht vollzogen werden. Da der Schnittstellenzugang jedoch unabhängig von der Grammatik der Anfragesprache ist, können Änderungen der Schnittstellenfunktionalität ohne Anpassung der Zugangsfunktionen des Nutzers (Debugger) vorgenommen werden. Sollte die Grammatik geändert werden, so muß die durch sie beschriebene Sprache die alte Anfragesprache vollständig enthalten. Somit sind grundlegende Änderungen der TDI-Funktionalität ohne Änderungen an der Nutzersoftware (Debugger) möglich.

Optionalität

Ein weiterer Gedanke bei der Entwicklung des TDI war seine Optionalität. Es soll möglich sein, das TDI nur dann in die Pthreads-Laufzeitumgebung zu integrieren, wenn es für das Debugging notwendig ist. Diese Einbindung könnte sogar erst zur Programmlaufzeit durch dynamisches Laden erfolgen. Für den Nutzer ist es damit nicht mehr zwingend notwendig, die Erzeugung der fertigen Applikation abhängig vom Debugging durchzuführen. Um ein Thread-Debugging für POSIX-Threads auf Basis des TDI's zu ermöglichen, muß keine Testversion der Applikation erzeugt werden. Ist das Laden des TDI zur Laufzeit nicht möglich, so sollte alternativ eine Einbindung zur Compile-Zeit möglich sein. Beide Formen der Einbindung des TDI werden in Abschnitt 7.2.2.1 beschrieben.

Das Laden und Aktivieren des TDI muß durch die POSIX-Threads-Implementierung erfolgen. Gesteuert wird das Laden des TDI durch den Debugger. Er muß der Pthreads-Laufzeitumgebung mitteilen, ob ein Pthreads-Debugging gewünscht ist.

Flexibilität

Die Erbringung der Debug-Funktionalität wird in starkem Maße von der Leistungsfähigkeit der Kapselungsschicht (TED) der jeweiligen Pthreads-Implementierung abhängig sein. Das TDI muß sich an die jeweilige TED-Implementierung anpassen. Sollte in der Kapselungsschicht nicht die gesamte Funktionalität für den Zugriff auf die Pthreads-Objekte bereitgestellt werden können, so muß das TDI trotzdem lauffähig sein.

Diese Anpassungsfähigkeit des TDI soll es ermöglichen, auch ältere Pthreads-Implementierungen zu unterstützen, die nicht mehr der neuesten TDI-Funktionalität entsprechen. Umgekehrt heißt das auch, daß neuere TDI-Versionen keine Anpassung der Pthreads-Implementierung nötig machen.

6. Schnittstellenentwurf

Die folgenden Abschnitte beschäftigen sich mit dem Entwurf des TDI, das heißt der Festlegung der Architektur der Thread-Debug-Schnittstelle unter Einhaltung der im vorigen Kapitel definierten Anforderungen.

Neben der Erbringung einer möglichst umfangreichen Debug-Unterstützung durch das TDI waren die in dieser Arbeit verfolgten Entwurfsziele vorrangig die Wartbarkeit, Erweiterbarkeit und Unabhängigkeit des TDI-Systems bei größtmöglichem Nutzen für das Pthreads-Debugging.

In den folgenden Abschnitten wird zunächst die Schnittstellenarchitektur als Ergebnis des Entwurfs vorgestellt. Die einzelnen Subsysteme werden dann separat betrachtet und erläutert.

6.1. TDI-Architektur

Das Gesamtsystem TDI kann wie in Abbildung 6.1 dargestellt in die Teilkomponenten **Kommunikation**, **Anfragebearbeitung** , **Verwaltungs- und Hilfsfunktionen** und **Pthread-Kapselung (TED)** dekomponiert werden.

Das Kommunikationsmodul ist für den Datenaustausch zwischen Debugger und TDI zuständig und ist der für den Debugger oder allgemein für den Schnittstellennutzer sichtbare Teil.

Alle Anfragen des Debuggers werden in einer Anfragesprache formuliert, um der Forderung nach einer einfachen und erweiterbaren Form des Schnittstellenzugangs gerecht zu werden. Um die Anfragen auszuwerten, müssen diese von einem Parser auf ihre syntaktische Korrektheit überprüft werden. Die der Grammatik der Anfragesprache entsprechenden Befehle werden von der Anfrageauswertung evaluiert. Parser und Anfrageauswertung werden zusammengefaßt als Anfragebearbeitung bezeichnet.

Die für die Anfragebearbeitung notwendigen Zugriffsfunktionen für die Laufzeitumgebung der POSIX-Threads-Implementierung werden vom TED[1]-Modul zur Verfügung gestellt. Dieses gewährleistet einen uniformen Zugriff auf alle für das Thread-Debugging relevanten Daten der Pthread-Implementierung.

[1] Thread-Erweiterung für das Debuging

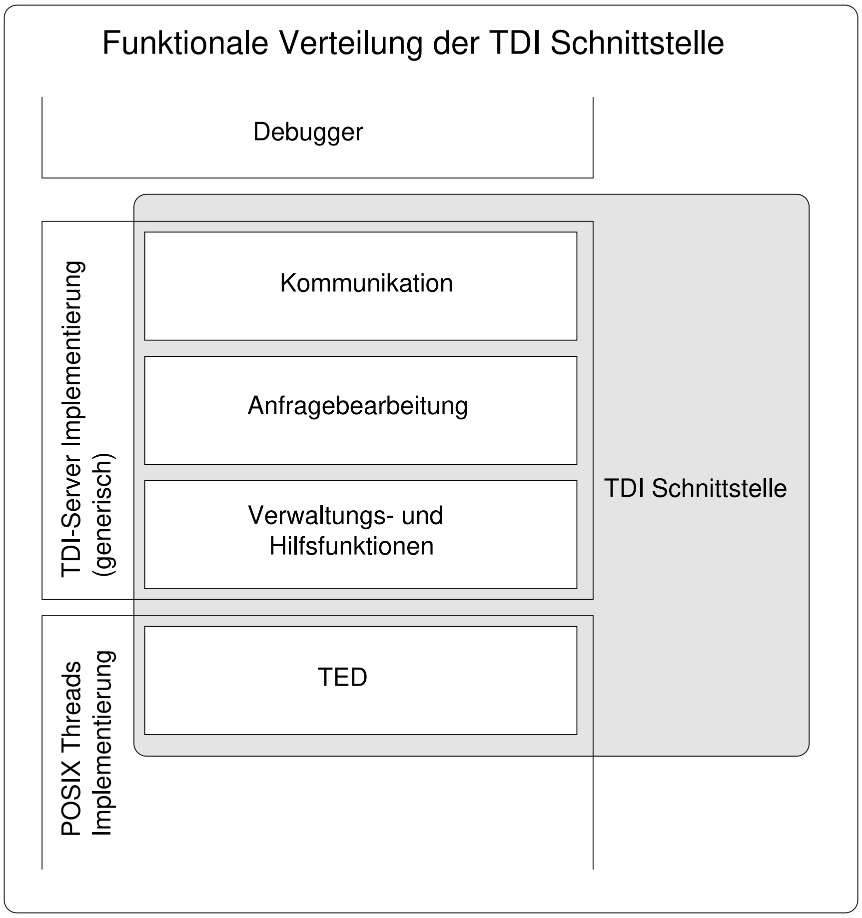

Abbildung 6.1.: Die TDI-Subsysteme

Da einige Verwaltungs- und Hilfsaufgaben keiner der bisher genannten Komponenten zugeordnet werden können, werden diese in dem Modul Verwaltungs- und Hilfsfunktionen zusammengefaßt. Dazu gehören Aufgaben der Registrierung von Pthreads-Objekten, die nicht von den Implementierungen verwaltet werden, oder die Bereitstellung von persistenten ID's für diese Objekte.

6.2. TED - Eine Pthreads-Erweiterung für das Debugging

Die Grundlage für die Schaffung einer einheitlichen Thread-Debug-Schnittstelle ist die Kapselung der Details der verschiedenen POSIX-Threads-Implementierungen. Dazu wird jede Pthread-Implementierung um ein Kapselungsmodul erweitert, welches eine einheitliche Schnittstelle - TED - implementiert, die dem TDI-Kern bereitsteht (siehe Abbildung 6.2).

Für die Entwicklung des TDI bedeutet das Vorhandensein der TED-Kapselungsschicht eine Aufwandsminimierung, da es generisch implementiert werden kann. Um Änderungen an bestehenden Pthread-Paketen zu vermeiden, könnte das TDI auch multifunktionalen Charakter haben, wie es in Abbildung 6.2 schematisch dargestellt ist. In diesem Fall müßten dem TDI die Details aller Implementierungen bekannt sein, um die für das Thread Debugging relevanten Daten zu extrahieren.

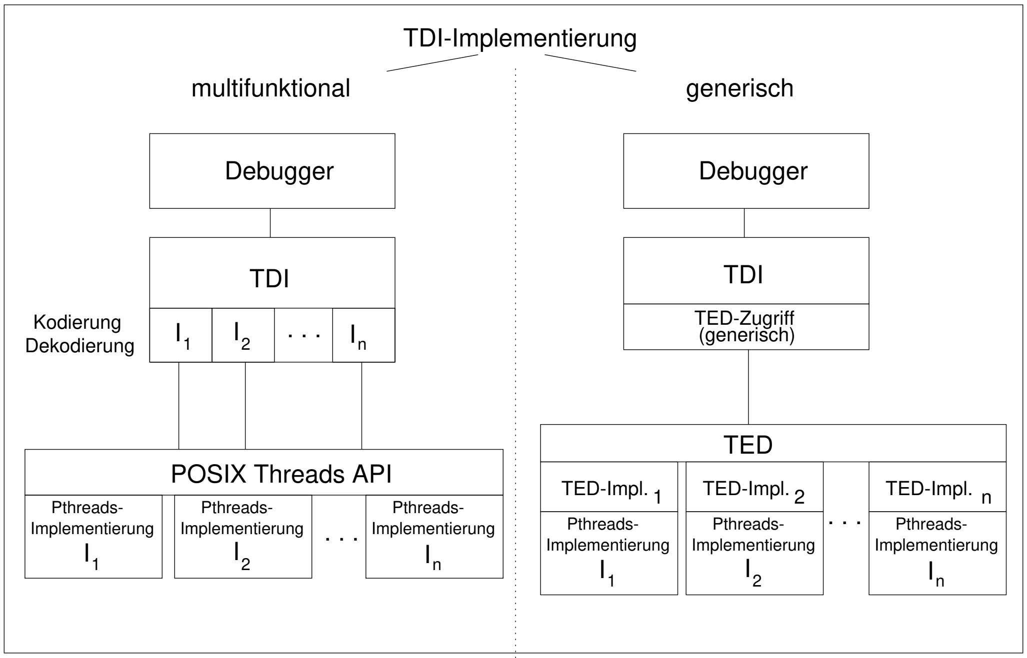

Abbildung 6.2.: Möglichkeiten der Kapselung

Bei gleicher TDI-Funktionalität stiege die Komplexität mit der Anzahl der zu unterstützenden Implementierungen. Die Entwicklung des TDI mit der geforderten Funktionalität ohne die Integration von Teilfunktionalität in die POSIX-Threads-Implementierung ist zudem mit sehr hohem Aufwand verbunden, da dem TDI nur das POSIX-Threads-API zur Verfügung stünde. Objekte mit standardisiertem Typnamen (z.B. `pthread_t, pthread_cond_t` und `pthread_mutex_t`) könnten aufgrund ihrer nicht standardisierten Struktur nur mit detaillierten Informationen über den internen Aufbau der Datenstrukturen gelesen und geschrieben werden. Beispielsweise müßten für alle implementierungsspezifischen Ausprägungen des Datentyps `pthread_t` Zugriffsroutinen implementiert werden, die dessen Struktur in eine Normalform transformierten. Ein weiterer Vorteil der Kapselungsschicht ist die Skalierbarkeit der Thread-Debug-Schnittstelle ohne Erweiterung der Implementierung, da für neue POSIX-Thread-Implementierungen nur die entsprechenden Kapselungsmodule zur Verfügung gestellt werden müssen.

In dem in Abschnitt 4.2 vorgestellten *SmartGDB* wird der multifunktionale Ansatz verfolgt. Er zeichnet sich durch eine schlechte Erweiterbarkeit, ungenügende Portabilität und die kurze Gültigkeit aus. Die Unterstützung eines neuen Pthreads-

Paketes ist immer mit einer Änderung der *SmartGDB*-Implementierung verbunden. Bei Änderungen an bereits vom *SmartGDB* unterstützten Pthreads-Implementierungen muß dieser ebenfalls modifiziert werden, da die Dekodierungsfunktionen des *SmartGDB* auf den internen Strukturen der Pthreads-Implementierung arbeiten. Diese Dekodierung ist weiterhin nur mit erweiterten Symbolinformationen (Compiler-Option „-g") des Programmes möglich.

In der hier vorgestellten Lösung wird der Zugriff auf die Objekte des Pthreads-Standards seitens der Pthreads-Implementierung durchgeführt. Das TED-Modul hat die Aufgabe, Objekte der Pthreads sichtbar zu machen und einheitliche Zugriffsmethoden zur Verfügung zu stellen. Dazu müssen je nach Implementierung der POSIX-Threads mehr oder weniger viele Berechnungen auf internen Datenstrukturen vollzogen werden, um die vorhandenen Daten in eine einheitliche Repräsentation zu transformieren. Diese implementierungsspezifischen Teile des TDI müssen für die Einhaltung des generischen Ansatzes in die POSIX-Thread-Implementierung verlagert werden; andere allgemeine Aufgaben können vom TDI-Kern erfüllt werden.

6.2.1. Objekte der TED

Die drei grundlegenden im POSIX-Threads-Standard definierten Objektklassen sind Thread, Mutex und Bedingungsvariable. Weitere Klassen wie Scheduler oder Implementierungsprofil könnten einbezogen werden, spielen aber im Zusammenhang mit dem Thread Debugging eine untergeordnete Rolle.

Objektmengen

Die zur Laufzeit instantiierten TED-Objekte müssen für den Nutzer sichtbar gemacht werden. Eine damit verbundene wichtige Forderung an das TED ist die Bereitstellung von Mengeniteratoren, die nicht Bestandteil des POSIX-Threads-Standards sind. Das TED bietet einen einfachen Mechanismus für den Mengenzugriff an, der aus den Primitiven `getFirst` und `getNext` für den Zugriff auf eine einfach verkettete Liste besteht.

Für das Scheduling in Bibliotheksimplementierungen werden Threads in der Regel schon in Listen gehalten, sodaß die Realisierung von Iteratoren unproblematisch ist. Die Bereitstellung des Mengenzugriffs für Mutexe und Bedingungsvariablen gestaltet sich schwierig, da diese im allgemeinen nicht in globalen Datenstrukturen gehalten werden. Im Abschnitt 6.5 wird ein genereller Lösungsansatz vorgestellt. Dieser basiert auf der Registrierung von Mutexen und Bedingungsvariablen beim TDI.

Attribute

Neben der Mengeniteration muß auch ein Zugriff auf die Attributwerte der einzelnen Objekte möglich sein. Attribute für Threads sind zum Beispiel die Priorität, der derzeitige Ausführungszustand oder die Startfunktion. Der POSIX Standard definiert für jede der Objektklassen spezielle Attributcontainer `pthread_...attr` , auf die ein Zugriff ausschließlich durch definierte Attributzugriffsfunktionen der Form `pthread_...attr_[get/set]<Attribute>` möglich ist. Ein Vorschlag in diesem Zusammenhang könnte lauten, den Zugriff auf die im Standard definierten Objekte, das heißt auf die Objektmenge und die einzelnen Objektattribute, über die Attributstrukturen zu realisieren. Dagegen spricht, daß es keine bijektive Abbildung zwischen der Menge der instantiierten Objekte und der Menge der Attributstrukturen gibt, die einem Pthread-Objekt eindeutig eine Attributstruktur zuordnet. Zum einen muß nicht zu jedem Pthread-Objekt eine Attributstruktur angelegt werden. In diesem Fall werden Vorgabewerte benutzt. Andererseits kann ein Attribut mehreren Pthread-Objekten zugeordnet sein. Diese letzte Möglichkeit wird sehr häufig benutzt, wenn mehrere Threads mit den gleichen Eigenschaften versehen werden sollen. Abbildung 6.3 zeigt mögliche Zuordnungen zwischen Threads und Attributen.

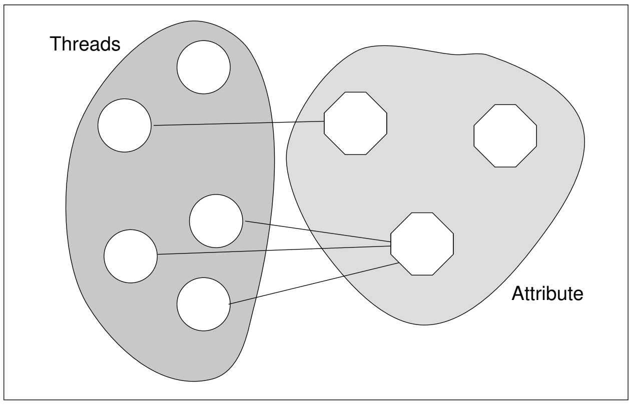

Abbildung 6.3.: Mögliche Zuordnung von Thread- zu Attributobjekten

Da nicht zu jedem Pthread-Objekt eine Attributstruktur vorhanden sein muß, kann der Attributzugriff nicht mittels POSIX-API realisiert werden. Im TED müssen deshalb Funktionen bereitgestellt werden, die einzelne Attribute eines bestimmten Threads lesen und wenn möglich schreiben können.

Vereinfachend wird definiert, daß alle Attribute eine einheitliche skalare Wertedomäne D_A besitzen, die Obermenge des Speicheradreßraumes sein muß. Beim Hinzufügen von neuen Attributen stellt diese Definition keine Einschränkung dar. Sind Attribute nicht skalar oder ist deren Wertedomäne nicht Teilmenge von D_A, so ist sichergestellt, daß zumindest die Speicheradresse dargestellt werden kann. Ein Zugriff auf den Attributwert muß dann mit Hilfe des Debuggers geschehen.

Die Beschränkung auf eine Domäne für alle Attribute ermöglicht die Festlegung einer einheitlichen Signatur für alle Attributlesefunktionen

$$S_r : T_{D_O} \to T_{D_A}$$

und für Schreibfunktionen

$$S_w : T_{D_A} \times T_{D_O} \to T_{D_A},$$

mit den Wertedomänen $dom(T_{D_A}) = D_A$ und $dom(T_{D_O}) = D_O$. D_O ist die Wertedomäne für die Adressen der Pthread-Objekte. Da die Wertedomäne D_A die Menge der Objektadressen für Thread, Mutex und Bedingungsvariable enthält, kann in den angegebenen Signaturen D_O durch D_A ersetzt werden. Diese einheitliche Festlegung der Signaturen ermöglicht eine einfache Realisierung des Anfragemechanismus sowie eine problemlose Erweiterbarkeit des TDI.

Jede Attributlesefunktion ist einem Attribut zugeordnet und liest den Attributwert eines Objektes. Eine Schreibfunktion schreibt den neuen Attributwert und gibt das alte Datum zurück. Dies ist besonders hilfreich, wenn einmal zugewiesene Attribute wieder auf ihren alten Wert gesetzt werden sollen, da man sich das Auslesen des alten Wertes vor der Schreiboperation spart.

Kodierung von Attributwerten

Der Nachteil der Definition von einheitlichen Signaturen für die Attributlese- und Schreibfunktionen ist die schwache Typisierung der Paremeter und Rückgabewerte, da diese den universellen Typ T_{D_A} besitzen. Deshalb ist es erforderlich, Kodierungsrichtlinien für Parameter und Rückgabewerte zu spezifizieren. Diese Kodierungsrichtlinien müssen bei der Implementierung der Attributzugriffsfunktionen beachtet werden. Für die im POSIX-Standard definierten Attribute werden deren Werte direkt gelesen und zurückgegeben. Attribute, die ausschließlich im TED definiert sind, müssen den TED-Kodierungsrichtlinien in Anhang A folgen.

Die Problematik der Wertekodierung soll am Beispiel des Threadattributes `state` erläutert werden. Bei `state` handelt es sich um ein synthetisches TED-Attribut, da der POSIX-Standard keine Definition des Threadzustandes vornimmt. Die Definition des Wertebereiches von `state` setzt die Festlegung eines Thread-Zustandsmodells voraus. Hier wird definiert, daß ein Thread im Laufe seiner Existenz die Grundzustände

- `RUNNING` - Thread ist aktiv,
- `READY` - Thread ist ausführungsbereit und wartet auf CPU-Freigabe,
- `BLOCKED` - Thread ist blockiert, und
- `EXITING` - Thread-Ausführung ist beendet

einnehmen kann. Abbildung 6.4 zeigt das Zustandsüberführungsdiagramm. Da der Zustand `BLOCKED` verschiedene Ursachen haben kann, die für den Programmierer interessant sind, wird er nochmals in die Subzustände $\texttt{BLOCKED}_{\texttt{Mutex}}$, $\texttt{BLOCKED}_{\texttt{Cond}}$, $\texttt{BLOCKED}_{\texttt{Timer}}$, $\texttt{BLOCKED}_{\texttt{Suspend}}$ und $\texttt{BLOCKED}_{\texttt{Other}}$ unterteilt.

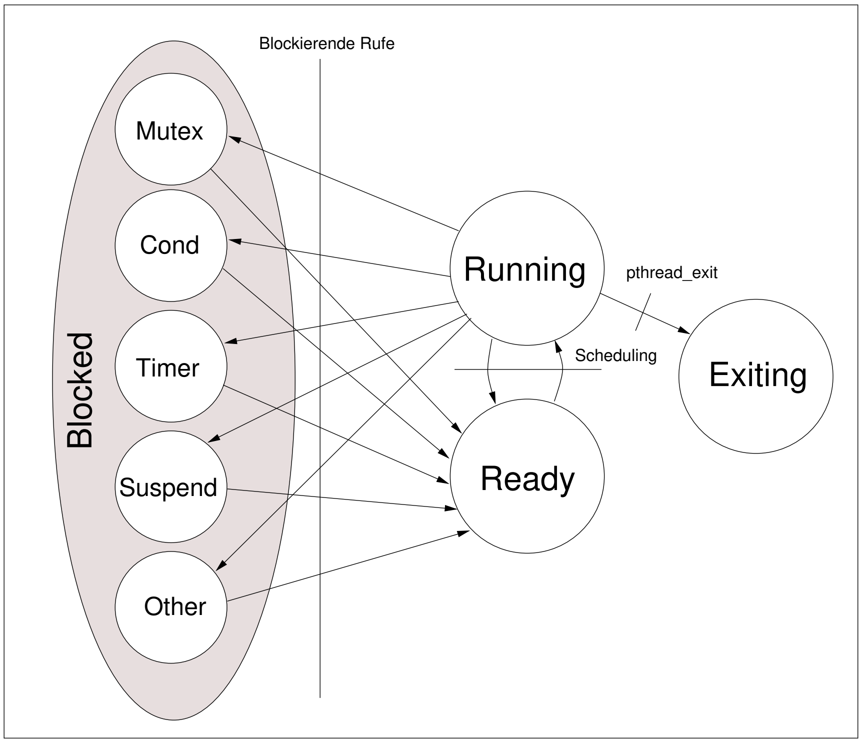

Abbildung 6.4.: Threadzustände

Verwendung weiterer Attributzugriffsmethoden

Abschließend muß noch hinzugefügt werden, daß die gesamte Thread-Debug-Unterstützung des TDI durch das Lesen und Setzen von Objektattributen erreicht werden kann. Es werden also nur die Attributzugriffsmethoden der Pthreads-Objekte benutzt. Die Philosophie dabei ist, daß nur die Werte, die für den Nutzer der TDI-Schnittstelle sichtbar sind, auch verändert werden können.

Im Vorfeld dieser Arbeit wurden im Rahmen der Erstellung eines Thread-Debugger-Prototypen für die FSU-Pthreads [Sch98] Befehle zur Beeinflussung des Scheduling bereitgestellt, die einen Thread suspendieren und einen suspendierten Thread wieder zur Ausführung bringen können. Eine weitere Funktion gestattet es, einem bestimmten nicht aktiven Thread die Ausführung zu ermöglichen. Die Funktionen hatten folgende Signatur:

`void pthread_suspend_th (pthread_t t)`	- Suspendieren des Threads t,
`void pthread_resume_th (pthread_t t)`	- Reintegration des Threads t und
`void pthread_switch_th (pthread_t t)`	- Aktivieren des Threads t.

Übertragen in die objektorientierte Welt könnte man die Funktionen auch als Methoden der Klasse `Thread` betrachten. Neben den beschriebenen trivialen Methoden zum Lesen und Schreiben der Attribute[2] der POSIX-Thread-Obkjekte wären somit weitere denkbar, mit denen komplexere Zustandsüberführungen möglich sind. Die angesprochenen Methoden zur Thread-Kontrolle könnten jedoch auch realisiert werden, indem das Attribut `state` verändert wird. Somit könnten wieder die allgemeinen Attributzugriffsfunktionen benutzt werden. Die Funktion `pthread_suspend_th` könnte durch Schreiben des Attributes `state` mit dem Wert $\texttt{BLOCKED}_{\texttt{Suspend}}$ realisiert werden, wobei der alte Wert von `state` als Ergebnis der Schreiboperation zurückgegeben wird. Ein `pthread_resume_th` wird realisiert, indem der alte Wert von `state` erneut zugewiesen wird. `pthread_switch_th` ist mit der Zuweisung `state = RUNNING` zu realisieren. Diese Zuweisungen des Attribues `state` bedeuten für den Nutzer eine elegante Form des Eingriffs in das Scheduling. Sie sind jedoch auf der Seite der Pthreads-Implementierung mit erheblichem Impelementierungsaufwand verbunden, da von der TED-Implementierung neben der Zuweisung des Attributes `state` auch komplexere Operationen auf den Warteschlangen der Threads ausgeführt werden müssen.

6.2.2. Objektbeziehungen

Neben den Objekteigenschaften, die direkt oder indirekt im API des POSIX-Thread-Standards definiert werden, gibt es auch standardinhärente Beziehungen zwischen den definierten Objekt-Klassen. Ist beispielsweise ein Thread T blockiert, so kann man dies durch Lesen des Attributes `state` feststellen. Ist T blockiert, weil er versucht, in den Besitz eines bereits vergebenen Mutex M zu gelangen, so ist das Tupel $[T, M]$ Element der Relation `BlockedOn`.

Für das Debugging einer POSIX-Threads-Applikation ist es von grundlegender Bedeutung, alle wichtigen im POSIX-Standard implizit oder explizit definierten Objektbeziehungen sichtbar zu machen. Diese Beziehungen sind im Klassendiagramm [Boo94] in Abbildung 6.5 dargestellt und sollen im folgenden näher betrachtet werden.

6.2.2.1. Beziehungen zwischen Thread und Mutex

Thread und Mutex können zur Laufzeit in zweifacher Hinsicht in Beziehung stehen:

i) Threads besitzen Mutexe (**OwnedBy**) oder

ii) Threads blockieren auf Mutexe (**BlockedOn**).

[2] Die Attributzugriffsfunktionen werden hier als Methoden der jeweiligen Objektklasse bezeichnet.

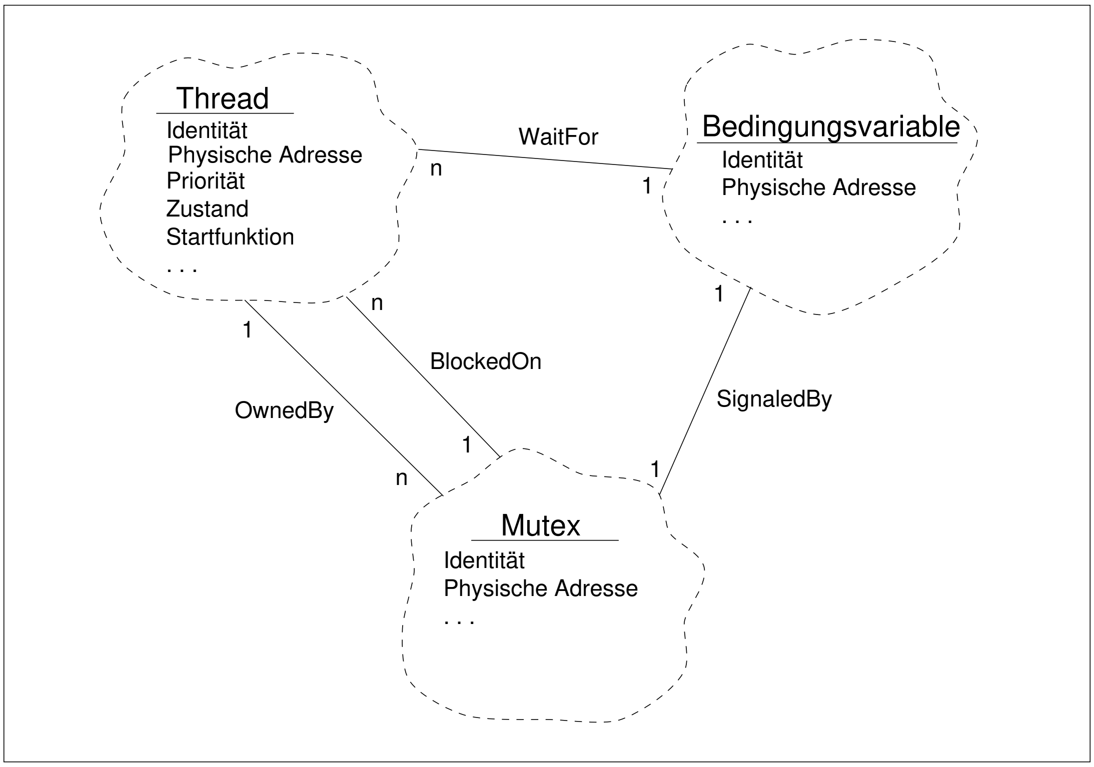

Abbildung 6.5.: Klassendiagramm der POSIX-Threads Objektklassen

Sei M die Menge der derzeit instantiierten Mutexe und T die Menge der in der Laufzeitumgebung befindlichen Threads. Für M gibt es eine injektive Abbildung $ID_M : M \to D_M$, die jedem Mutex eine ID aus der Domäne D_M zuordnet, und für die Menge T existiert dementsprechend die Abbildung $ID_T : T \to D_T$, die jedem Thread eine ID aus der Domäne D_T zuordnet. Für die in **i)** erwähnte Beziehung wird die Relation **OwnedBy** mit dem Schema **OwnedBy**{ThreadID:D_T, MutexID:D_M} eingeführt [Vos94]. Abbildung 6.5 zeigt die Darstellung von **OwnedBy** im Booch-Klassendiagramm. Die Kardinalität der Relation ist 1:N, das heißt ein Thread kann mehrere Mutexe besitzen, wobei jeder Mutex laut Mutex-Definition nur mit einem Thread in der **OwnedBy**-Beziehung stehen kann. Die Umwandlung des Booch-Modells in das Relationenmodell kann vereinfacht werden, indem der Relation Mutex das Attribut `Owner` hinzugefügt wird. Damit stellt das Hinzufügen des Attributs Owner zur Relation Mutex die einzige Änderung des Relationen-Modells der POSIX-Objektklassen für die Umsetzung der Beziehung **OwnedBy** dar.

Für die Möglichkeit **ii)**, bei der ein Mutex und die auf ihn blockierten Threads betrachtet werden, ergibt sich ebenfalls die Kardinalität 1:N. Ein Mutex kann die Ursache für ein blockierendes `pthread_mutex_lock` mehrerer Threads sein; ein Thread kann jedoch nur auf einen Mutex blockieren, da die Ausführung eines

Threads sequentiell ist und durch das POSIX-API nur die Möglichkeit zur Anforderung eines Mutex bereitgestellt wird. Aufgrund dieser Kardinalität kann die Booch-Darstellung der Relation mit dem Schema **BlockedOn** $\{$`ThreadID:`D_T, `MutexID:`$D_M\}$ in das Relationenmodell umgewandelt werden, indem der Relation Thread das Attribut `MutexBlockedOn` hinzugefügt wird.

6.2.2.2. Beziehungen zwischen Threads und Bedingungsvariablen

Für den Entwickler von Pthreads-Applikationen ergibt sich eine weitere wichtige Beziehung zwischen Threads und Bedingungsvariablen, die **WaitFor**-Beziehung. Ein Thread T und eine Bedingungsvariable CV bilden ein Tupel der Relation **WaitFor**, wenn Thread T bei der Ausführung von `pthread_cond_wait(CV,M)` blockiert, wobei M der mit der Bedingungsvariable CV assoziierte Mutex ist. Die Kardinalität der Beziehung **WaitFor** ist 1:N. Das heißt, ein Thread kann nur auf die Signalisierung einer Bedingungsvariable warten; eine Bedingungsvariable hingegen kann der Grund für das Blockieren mehrerer Threads sein. Der naive Ansatz, eine Relation mit dem Schema **WaitFor** $\{$`ThreadID:`D_T, `CondVarID:`$D_{CV}\}$ einzuführen, kann durch die Kenntnis der Kardinalität verworfen werden, und anstelle der Einführung einer neuen Relation wird das Schema der Relation Thread um das Attribut `CVWaitingFor` erweitert.

Im Gegensatz zur Beziehung **OwnedBy** zwischen Thread und Mutex wird hier keine Beziehung **Signaled** definiert, die die Signalisierung eines Threads für eine bestimmte Bedingungsvariable ausdrückt, da es sich hierbei um ein Ereignis und nicht um einen Zustand handelt.

6.2.2.3. Beziehungen zwischen Mutex und Bedingungsvariable

Mit der Ausführung der Funktion `pthread_cond_wait(CV,M)` bilden die Bedingungsvariable CV und der Mutex M ein Tupel der Relation **SignaledBy**. Die Kardinalität der Beziehung **SignaledBy** ist laut POSIX Standard 1:1. In einigen Standard-Implementierungen (z.B. MIT-Pthreads [PHR93]) sind jedoch andere unzulässige Kardinalitäten möglich. Standardgemäß ist es, wenn nur eine Bedingungsvariable CV für die Signalisierung eines Mutex M benutzt wird. Eine Bedingungsvariable kann nicht für mehrere Mutexe verwendet werden.

6.2.2.4. Erweiterung des Modells

Bisher wurde gezeigt, daß alle Objektbeziehungen durch zusätzliche Attribute der im TED sichtbaren Objekte dargestellt werden können. Sollten durch Standardänderungen oder bisher nicht betrachtete Objektbeziehungen M:N-Kardinalitäten auftreten, so müssen für diese Beziehungen Objektklassen eingeführt werden, da nur skalare Attribute der Domäne D_A (siehe Abschnitt 6.2.1) zugelassen sind.

Diese Klassen dienen ausschließlich der Repräsentation der Objektbeziehung und besitzen deshalb als Attribute die Schlüssel der beteiligten Relationen.

6.3. TED-Zugriff durch das TDI

Mit der Thread-Erweiterung der Pthreads für das Debugging (TED) steht der Basisbefehlssatz zur Verfügung, um die Objekte der POSIX-Threads sichtbar zu machen. In Abschnitt 5.1 wurde die Anforderung spezifiziert, eine möglichst einfache anfrageorientierte Schnittstelle zwischen Debugger und Pthreads (beziehungsweise TED) bereitzustellen. Diese Forderung ist zum einen aus Entwicklersicht nachzuvollziehen, andererseits ergeben sich durch eine kompakte und variable Anfragesprache mehr Möglichkeiten für den Benutzer. Die Abarbeitung von Sammelanfragen durch das TDI (deklarativer Ansatz) ist zudem effektiver als das Aufrufen jeder einzelnen im TED bereitgestellten Funktion (prozeduraler Ansatz) durch den Debugger.

Der folgende Abschnitt beschäftigt sich mit Fragen der Gewinnung und Haltung von Pthreads-Daten und mit dem Entwurf einer Anfragesprache.

6.3.1. Datengewinnung - Redundante Ausführung vs. Redundante Speicherung

Betrachtet man eine Multithread-Applikation zu einem bestimmten Zeitpunkt, so stellen die instantiierten Objekte der in Abschnitt 6.2.1 vorgestellten Objekttypen die Extension einer relationalen Datenbank mit den in Abschnitt 6.2.1 definierten und in Abschnitt 6.2.2 erweiterten Relationen dar. Die Abbildung 6.6 skizziert eine Beispielsituation. Thread A und Thread B führen nebenläufig das dargestellte Programmstück aus, in dem ein kritischer Abschnitt durch den Mutex `MUTEX` geschützt ist. Im kritischen Abschnitt ist ein Breakpoint gesetzt, der von Thread A erreicht wird. Ein Auslesen der zu diesem Zeitpunkt vorhandenen Tupel der Relationen **Thread**, **Mutex** und **Bedingungsvariable** führte zu dem in den Tabellen dargestellten Ergebnis.

Bei bisherigen Thread-Debugger-Lösungen kann festgestellt werden, daß durch den Debugger eine Protokollierung der Ereignisse der Laufzeitumgebung durchgeführt wird. Dabei werden auf der Seite des Debuggers die in Abbildung 6.6 dargestellten Tabellen schrittweise aktualisiert. Das hat den Vorteil, daß eine Nutzeranfrage direkt vom Debugger ohne Ausführung von Teilen der Applikation beantwortet werden kann. Neben der Gefahr der Inkonsistenz der gespeicherten Daten ist diese Lösung im Hinblick auf die Menge der extrahierbaren Daten sehr eingeschränkt. Eine Darstellung der **Mutex**- bzw. der **Bedingungsvariable**-Relation wäre nur unter erheblichem Dekodieraufwand der internen Datenstrukturen der Applikation möglich, da die Mutex- und CV-Strukturen nicht von der

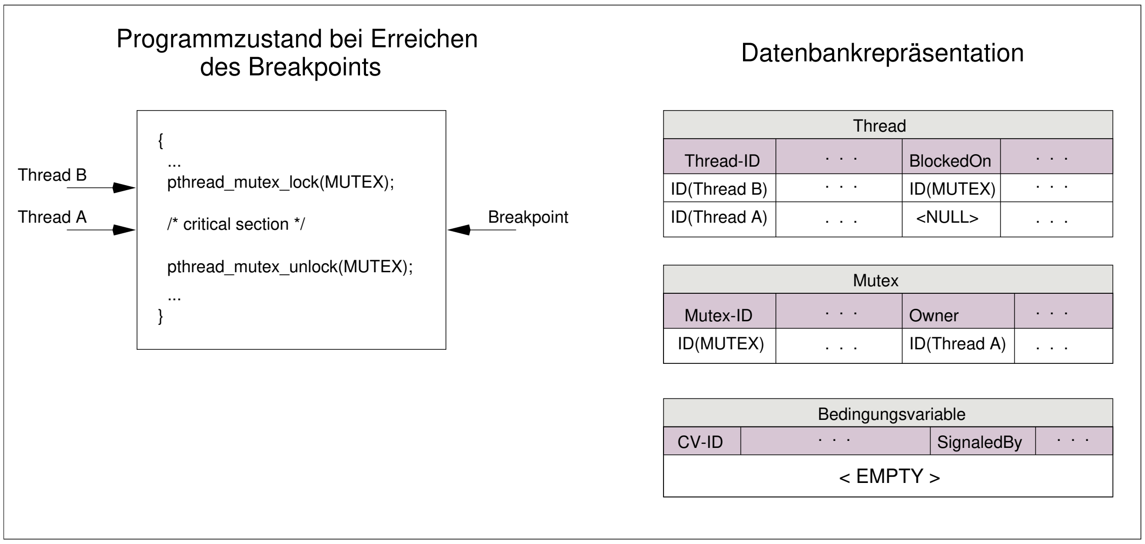

Abbildung 6.6.: Datenbankrepräsentation eines Programmzustandes

Pthread-Implementierung verwaltet werden[3]. Im allgemeinen ist eine redundante Speicherung der ohnehin vorhandenen Programmdaten auf der Seite des Debuggers fragwürdig.

In dieser Arbeit wird der Ansatz der aktiven Debug-Unterstützung durch die Multithread-Applikation verfolgt. Ein Protokollieren der Daten wird ersetzt durch das Auslesen der im Speicherabbild der Applikation vorhandenen Daten durch die TED-Funktionen. Eine Datenhaltung der Antwortdaten findet nur temporär statt. Der vermeintliche Mehraufwand des wiederholten Auslesens steht dabei dem Aktualisierungsaufwand und der geringen Leistungsfähigkeit der konventionellen Lösung gegenüber.

6.3.2. Anfragemöglichkeiten

Eine Übertragung der gesamten Tupelmenge aller vorhandenen Relationen würde zu einem hohen Laufzeit- und Speicherverbrauch für die Thread-Debug-Unterstützung führen. Aus diesem Grund ist es notwendig, nur die vom TDI-Nutzer benötigten Daten zu übermitteln. Um eine Reduzierung der Datenmenge zu erreichen, werden im TDI zwei Anfrageoperatoren der Relationalen Algebra implementiert [Cod70].

[3]Die vom Nutzer angelegten Mutexe und Bedingungsvariablen sind der Pthreads-Laufzeitumgebung nur zum Zeitpunkt der Ausführung der API-Funktionen bekannt.

6.3.2.1. Selektion

Häufig benötigte Informationen im Zusammenhang mit dem Thread-Debugging sind beispielsweise die ID des derzeit aktiven Threads oder der von ihm belegten Mutexe. Um diese Daten zu gewinnen, ist es für die erstgenannte Anfrage nötig, über die gesamte Tupelmenge der Thread-Relation zu iterieren und die Thread-Tupel zu suchen, bei denen das Attribut `state` den Wert `RUNNING` hat. Für die Gewinnung der Tupel der Mutexe, die von einem Thread T belegt sind, wird in der Mutex-Relation nach Tupeln gesucht, deren Attribut `Owner` den Wert $id_T(T)$ besitzt. Diese Operationen können als Auswahl bestimmter Tupel einer Relation durch ein Auswahlprädikat charakterisiert werden und entsprechen somit dem Selektions-Operator der Relationalen Algebra. Sei R eine Relation aus der Menge der m-stelligen Relationen R_m und $\Theta : R \to \{true, false\}$ ein Prädikat mit Attributbezeichnern von R als Eingabevariablen. Die Selektion σ_Θ auf der Menge R mit dem Prädikat Θ ist dann wie folgt definiert

$$\sigma_\Theta(R) := \{r \in R \mid \Theta(r)\}.$$

Die Selektion wird vom TDI angeboten und ist in der TDI-Anfragesprache formulierbar, deren Grammatik in Abschnitt 6.3.3 beschrieben ist.
Es ist noch anzumerken, daß die Selektion nicht die Ausführungszeit der Anfrage verringert, da für jedes Tupel von R die Evaluierung des Prädikatausdruckes erforderlich ist. Es sind jedoch Optimierungen möglich, die Resultat der Analyse des Prädikatausdruckes sein können. Bei der Anfrage $\sigma_{(ID==j)}(Thread)$ zum Beispiel könnte die Mengeniteration bei Auftreten des ersten Tupels mit erfülltem Prädikat $ID == j$ abgebrochen werden, da ID ein Schlüsselattribut der Relation $Thread$ ist.

6.3.2.2. Projektion

Für den TDI-Benutzer kann es sinnvoll sein, nur eine Teilmenge der Attribute einer Relation zu betrachten. Sollen bei allen Threads Breakpoints an der nächstmöglichen Stelle im Programmcode gesetzt werden, so sind nur die Werte des Threadattributes `newpc` erforderlich, um diesen Befehl zu implementieren. Ein Vorteil wäre, daß weniger Daten zwischen Applikation und Debugger übertragen werden müßten. Zudem wären im TED nur Aufrufe der Attributlesefunktion für `newpc` erforderlich. Damit entfallen die unter Umständen sehr laufzeitverbrauchenden Rufe für das Generieren und Lesen synthetischer Attribute (z.B `state`).

Die Projektion als zweite im TDI angebotene Grundoperation der relationalen Algebra hat die geforderte Semantik. Seien $D_1, D_2, \ldots, D_n$ Domänen und die Relation R eine Teilmenge aus $D_1 \times \ldots \times D_n$ mit dem Schema $R\{A_1 : D_1, \ldots, A_n : D_n\}$. Die Menge $Proj$ der projezierten Attribute sei $\{A_{j1}, \ldots, A_{jk}\}$. $\mu[S]$ oder auch $\mu[\{A_{j1}, \ldots, A_{jn}\}]$ bezeichnet die Werte des Tupels μ in den Attributen $A_{jk} \in S$.

Die Projektion $\Pi_{Proj}(R)$ ist definiert als

$$\Pi_{Proj}(R) := \{\mu[Proj] \mid \mu \in R\}.$$

Für das TDI kann vereinfacht werden, indem für alle Attribute die angesprochene Wertedomäne D_A angenomen wird.

Hinzuzufügen ist, daß das Ergebnis einer Operation der Relationalen Algebra immer eine Tupelmenge ist und demzufolge keine Duplikate enthalten sind. Das führt unter Umständen zu einer weiteren Reduktion der zu übertragenden Datenmenge. Die Duplikatentfernung muß bei der Formulierung der Anfrage berücksichtigt werden, stellt jedoch keine Einschränkung dar. Im obigen Beispiel ist es sinnvoll, die Menge und nicht die Liste der Attributwerte für `newpc` zu erhalten, da sonst Breakpoints mehrfach an der gleichen Programmadresse gesetzt würden.

6.3.2.3. Weitere Operatoren

Die vorgestellten Formulierungsmöglichkeiten Selektion und Projektion stellen die Grundlage für einen komfortablen Zugriff auf die Daten der TED dar. Die ausschließliche Unterstützung dieser Operationen stellt jedoch keine Einschränkung in Bezug auf die Erweiterbarkeit der Anfragemöglickeiten dar. Komplexere Anfragemöglichkeiten wie Mengenoperatoren und Kartesisches Produkt sowie darauf basierende Operatoren können auf der Seite des TDI-Nutzers implementiert werden. Für die in Kapitel 8 vorgestellten Thread-Debugging-Befehle sind die vorgestellten Operatoren ausreichend.

6.3.3. Die Anfragesprache TDI-QL

Die Anfragesprache des TDI soll die beschriebenen Möglichkeiten der Mengenadressierung, Projektion und Selektion bereitstellen. Neben dem Auslesen von Attributen soll es auch möglich sein, Attribute zu schreiben. Bei dem Entwurf der Anfragespache wurden Anleihen bei bekannten Anfragesprachen wie SQL[Ame89] oder SCPI[SCP97] gemacht.

Eine Anfrage `<TDI-QL Query>` besteht aus einer Zeile und hat die folgende BNF-Darstellung:

```
<TDI-QL Query>::= <Modus> <Anfrage>

<Anfrage>     ::= <TED-Anfrage>
                | <Erweiterte Anfrage>

<TED-Anfrage> ::= {'!'}<Relation>
                  ':'<Projektion und Zuweisung>
                  {':'<Selektion>}
```

Im Anfrageteil `Modus` kann spezifiziert werden, ob es sich bei der Anfrage um eine TED-Anfrage oder eine erweiterte Anfrage handelt. Das Anfragepräfix `Modus` bietet damit Möglichkeiten der Erweiterung der Anfragesprache, die im folgenden mit TDI-QL[4] bezeichnet wird. Beispielsweise wäre es vorstellbar, Befehle mit administrativem Charakter an die TDI-Schnittstelle zu übermitteln, die mit einem speziellen Präfix versehen werden. Der Modus-Bezeichner für TED-Anfragen ist „<TED>“ für Schreib- und Lesebefehle.

Mit <`Relation`> kann die Relation adressiert werden, für die die Anfrage bestimmt ist. Der Teil <`Projektion und Zuweisung`> der Anfrage gibt an, welche Attribute die Ergebnistupel beinhalten sollen. Dabei werden die Attribute als Liste der folgenden Form angegeben:

```
<Projektion und Zuweisung> ::= <Projektionsliste>

<Projektionsliste>         ::= <Projektionsliste> ',' <P-Listenelement>

<P-Listenelement>          ::= <Attributbezeichner> { '=' <DA-Wert> }

<DA-Wert>                  ::= <hexadezimal kodierter Attributwert>
```

Die Reihenfolge der Attributbezeichner ist dabei maßgeblich für die Ausgabereihenfolge. Damit wird dem Anfragenden die Möglichkeit gegeben, das Ausgabeformat des Ergebnisses zu bestimmen. Das Ergebnisformat ist wie folgt definiert:

```
<TDI-Antwort>   ::= <Tupelliste>
                    | <Fehlermeldung>

<Tupelliste>    ::= <Tupelliste> '#'
                    <Ergebnistupel>

<Ergebnistupel> ::= <Werteliste>

<Werteliste>    ::= <Werteliste> ' ' <DA-Wert+>

<DA-Wert+>      ::= <DA-Wert>
                    | '?' /* NULL */
```

Die Positionen der Werte in der <`Werteliste`> korrespondieren mit denen der Attributbezeichner in der Projektionsliste. Die Werte sind hexadezimal kodiert und Elemente der Menge $D_A \cup \{Null\}$. Dabei bezeichnet der Ergebniswert $Null$ den Umstand, daß das Attribut des betrachteten Tupels nicht gesetzt ist oder nicht gelesen werden kann. Bei Attributen, die mit Hilfe von `'='` geschrieben werden, erscheint der Wert im Ergebnistupel, der vor der Zuweisung vorhanden war.

[4] Thread Debug Interface Query Language

Da die Ergebnisse Mengencharakter haben, werden Duplikate eliminiert, wobei gleiche Tupel mit *Null*-Werten nicht als Duplikate betrachtet werden, da die Ursache für diesen Wert nicht definiert ist. Um auch die Duplikate in das Ergebnis aufzunehmen, das damit einen Listencharakter bekommt, bietet TDI-QL die Möglichkeit, dem Relationennamen ein '!' voranzustellen[5]. Zu beachten ist, daß nur der Mengencharakter des Ergebnisses spezifizierbar ist. Das Schreiben von Attributen geschieht in allen von der Selektion betroffenen Tupeln, auch wenn deren projezierte Teiltupel im Ergebnis zusammengefaßt werden.

Eine Anforderung im Hinblick auf die Flexibilität des TDI bestand darin, auch mit einer Teilmenge der im TED geforderten Funktionalität zu arbeiten. Können in einer TED-Erweiterung einer POSIX Threads-Implementierung nicht alle Attributzugriffsfunktionen zur Verfügung gestellt werden, so sollte das für eine Anfrage bestmögliche Ergebnis erzielt werden. Das TDI muß erkennen, welche Funktionalität der jeweiligen TED-Implementierung fehlt und dementsprechend das Ergebnis kodieren, indem die nichtlesbaren Attribute den *Null*-Wert erhalten.

Mit der Angabe des Prädikatausdruckes `<Selektion>` wird der Selektionsoperator auf die angegebene Relation angewendet. Die Bildungsvorschrift für den Prädikatausdruck sowie die gesamte Grammatik der Anfragesprache sind in Anhang B gegeben.

6.4. Kommunikation zwischen Debugger und TDI

Die Kommunikation zwischen dem Debugger und der Applikation besteht aus dem Übermitteln der Anfrage an das im Kontext der Applikation befindliche TDI und dem Transport des von der Schnittstelle berechneten Ergebnisses zum Debugger.

In den folgenden Abschnitten wird auf die in diesem Zusammenhang existierenden Besonderheiten und Probleme eingegangen. Augenmerk gilt dabei den für den Datentransfer bestehenden Einschränkungen aufgrund der besonderen Zustände, in denen sich die Kommunikationspartner zum Zeitpunkt des Datenaustauschs befinden. Diese Besonderheiten gelten jedoch nicht für Kernel-Thread-Implementierungen. Im Abschnitt 6.4.4 wird auf Kommunikationsmöglichkeiten für diese Pthreads-Lösungen eingegangen.

6.4.1. Einschränkungen für den Datenaustausch

Die hier betrachteten Betriebssysteme (UNIX-Derivate) bieten den folgenden Mechanismus für ein Breakpoint-Debugging an, der über die `ptrace`-Schnittstelle benutzt werden kann. Der Systemruf `ptrace` bietet die Möglichkeit, eine Reihe

[5] Vergleiche SQL-Schlüsselworte ALL und DISTINCT.

von Operationen auf dem Zielprozeß auszuführen [UNI, LMKQ88]. Diese Operationen umfassen sowohl das Lesen und Schreiben des Prozeßabbildes als auch die Ausführungskontrolle über den Prozeß.

Mit dem Systemruf `ptrace`$_{\texttt{POKE...}}$ sind beliebige Manipulationen im Prozeßabbild möglich. Beispielsweise können durch den Debugger sogenannte Trap-Instruktionen (Breakpoints) im Codesegment plaziert oder es kann der Ruf einer Funktion durch Umsetzen des `pc` und Schreiben der Argumente auf den Stack vorbereitet werden. Während der Ausführung der `ptrace`-Rufe ist der Zielprozeß gestoppt. Mit `ptrace`$_{\texttt{CONT}}$ wird die Ausführung des Zielprozesses fortgesetzt. Läuft der Zielprozeß auf eine Trap-Instruktion, so wird ein Software-Interrupt ausgelöst und der Prozeß wird gestoppt. Gleichzeitig kehrt der vom Debugger ausgeführte `wait`-Systemruf zurück. Die Abbildung 6.7 illustriert das beschriebene Szenario.

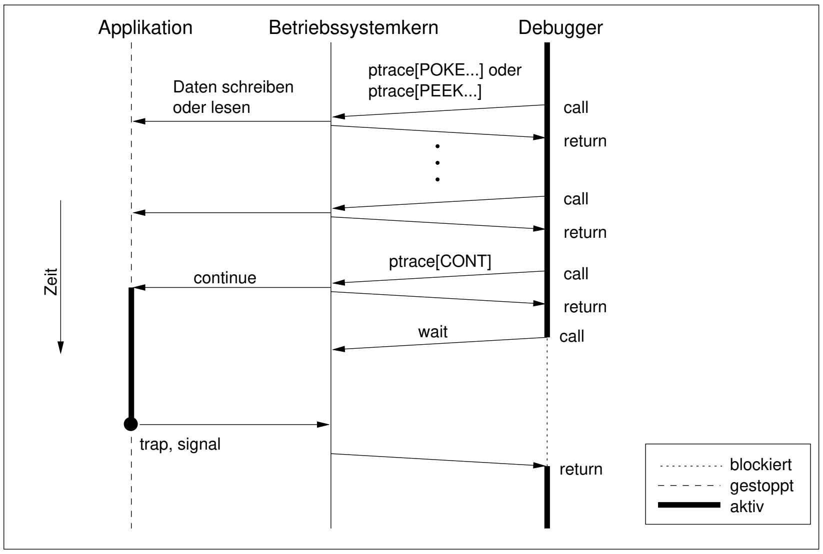

Abbildung 6.7.: Phasen des Breakpoint-Debugging

Betrachtet man die in der Abbildung 6.7 markierten Phasen, in denen die beiden Prozesse Debugger und Applikation aktiv sind, so stellt man fest, daß beide bis auf eine Synchronisationslatenz im gegenseitigen Ausschluß ausgeführt werden. Für den Datenaustausch zwischen Debugger und Applikation muß dieser Umstand der Synchronität berücksichtigt werden. Im folgenden sollen die Phasen der Kommunikation zwischen Debugger und TDI erläutert werden.

6.4.2. Phasen des Datenaustauschs

Es lassen sich zwei Phasen des Datenaustausches erkennen. In der ersten Phase benachrichtigt der Debugger die Applikation über die Art der gewünschten Information, indem er ihr einen Anfragesatz der Sprache **TDI-QL** übergibt. Die

Applikation bearbeitet diese Anfrage und übermittelt dann in der zweiten sogenannten Antwortphase die Ergebnisdaten an den Debugger. Die Bearbeitung der Anfragedaten sowie der Beginn der Antwortphase müssen durch eine Signalisierung der Applikation erfolgen, da sich der Zielprozeß im Stopp-Zustand befindet.

6.4.2.1. Anfragephase

Die Anfragephase ist dadurch charakterisiert, daß relativ zum möglichen Ergebnis kleine Datenmengen übertragen werden. Dementsprechend wurden nichtsymmetrische Lösungen für Anfrage- und Ergebnisphase erarbeitet.

Die Übertragung der Anfragedaten kann wie folgt realisiert werden. Anfangs etablieren Debugger[6] oder Zielapplikation einen Kommunikationskanal für die Interprozeß-Kommunikation (IPC)[Ste98]. Bei verbindungsorientierten IPC-Varianten wird die Realisierung erschwert, da beide Kommunikationspartner die Verbindung gleichzeitig etablieren müssen. Verbindungslose Varianten gestatten es, daß der jeweilige IPC-Kanal nur von jeweils einem Partner geöffnet werden muß, der dann Daten lesen oder schreiben kann.

Der Debugger als Anfragender schreibt seine Nachricht in den IPC-Kanal. Da ein gegenseitiger Ausschluß für die Ausführung von Sender und Empfänger vorliegt, wird die Anfrage noch nicht vom Empfänger gelesen. Einige IPC-Varianten (Message Queues) sind gut geeignet, Nachrichten zu schreiben, die erst zu einem späteren Zeitpunkt gelesen werden. Viele IPC-Formen bieten die Möglichkeiten der Pufferung (Message Queues, Pipes). Bei Anfragen, die länger als die internen Puffergrößen sind, muß die Nachricht zerteilt und mittels wechselseitigem Lesen und Schreiben der Nachrichtensegmente erfolgen. Für die Anfragephase kann jedoch vereinfachend vorausgesetzt werden, daß eine Überschreitung der Puffergröße aufgrund der Kompaktheit der Anfragen sehr unwahrscheinlich ist, wenn man die in den betrachteten Systemen verwendeten Puffergrößen verwendet. Der Datenaustausch in der Anfragephase kann daher mit einer einzigen Schreiboperation erfolgen.

6.4.2.2. Signalisierung

Nachdem der Debugger die Nachricht geschrieben hat, muß er die Applikation veranlassen, die Nachricht zu lesen. Dieser Vorgang wird im folgenden als Signalisierung bezeichnet. Im Falle einer User-Space-Implementierung erfolgt die Signalisierung, indem im Zielprozeß ein Funktionsruf vorbereitet und mit Hilfe von $\texttt{ptrace}_{\texttt{CONT}}$ initiiert wird. Die gerufene Funktion liest die Anfragedaten und

[6] Bei einigen IPC-Formen muß darauf geachtet, daß der Debugger als Vaterprozeß die Ressource öffnet, sodaß die Zielapplikation die zugehörigen Deskriptordaten erbt. Alle anderen IPC-Varianten können von Prozessen geöffnet werden, die zueinander nicht in der Eltern-Kind-Beziehung stehen.

übergibt sie dem Anfragemodul des TDI. Der Debugger führt nach dem Aufruf von `ptrace` ein blockierendes `wait` aus, bis die Antwortphase beendet ist.

6.4.2.3. Antwortphase

Nachdem die Anfrage vom TDI bearbeitet wurde, liegen im Ziel-Prozeß die Ergebnisdaten vor. Es wird davon ausgegangen, daß die Ergebnisdaten umfangreicher als die Anfragedaten sind und die Puffer der jeweils benutzten IPC-Variante nicht mehr ausreichen, um alle Daten mit einer Schreiboperation zu versenden. Die Lösung für die Übermittlung der Antwortdaten muß es gestatten, daß die Daten nach Beendigung der im Ziel-Prozeß ausgeführten Schreibroutinen vom Debugger lesbar sind, da er während der Ausführung dieser Funktionen blockiert ist und keine Daten lesen kann.

Eine Möglichkeit zur Lösung des Problems ist das puffergerechte Zerteilen der Ergebnisdaten unter wechselseitigem Schreiben des Ziel-Prozesses und Lesen des Debuggers (Abbildung 6.8). Der Speicher des Ziel-Prozesses fungiert dabei als Puf-

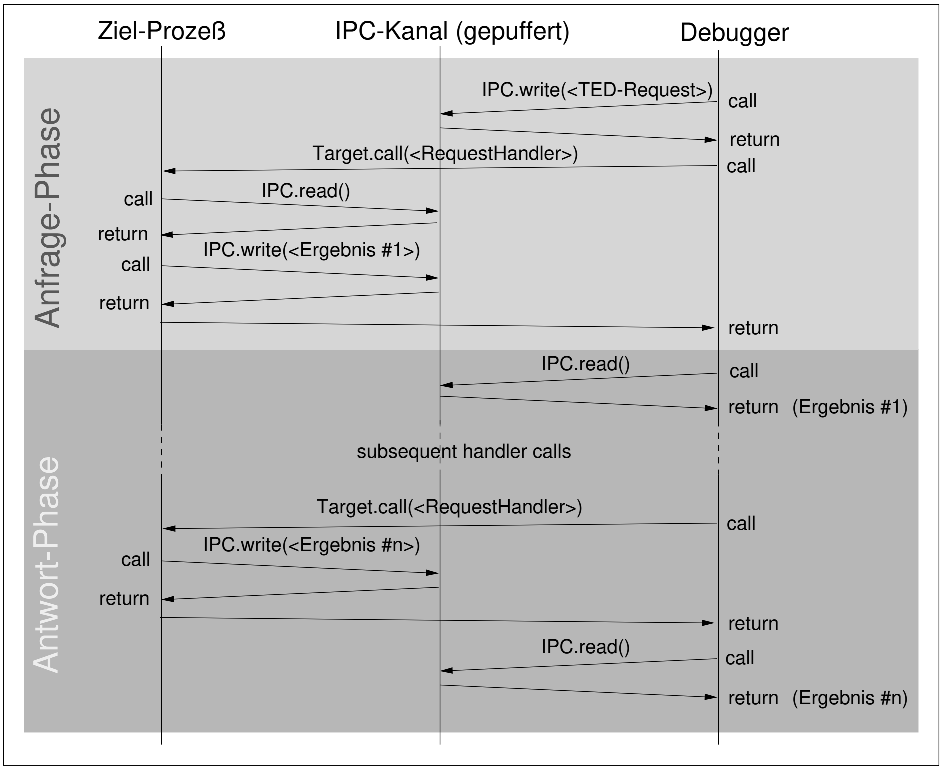

Abbildung 6.8.: Wechselseitiges Lesen und Schreiben

fer für alle gewonnenen Daten. Für jedes Schreiben eines Datenpaketes wird die Funktion `RequestHandler` im Ziel-Prozeß aufgerufen. Neben dem Mehraufwand für die Implementierung und der Erhöhung der Komplexität der Kommunikationsroutinen erfordert diese Variante eine laufzeitraubende Signalisierung.

Im folgenden wird die abgewandelte endgültige Lösung präsentiert, in der der Ziel-Prozeß nur einmal signalisiert werden muß. Die gewonnenen und im Speicher des Ziel-Prozesses abgelegten Daten sowie die benötigten Teile der Laufzeitumgebung des Prozesses werden dupliziert (mittels `fork`). Die Funktion, die vom Debugger zum Auswerten der Anfrage im Zielprozeß gerufen wurde, kehrt zurück und der duplizierte Zielprozeß kann den Datenaustausch mit dem nun wieder aktiven Debugger beginnen. Abbildung 6.9 skizziert die einzelnen Aktionen der

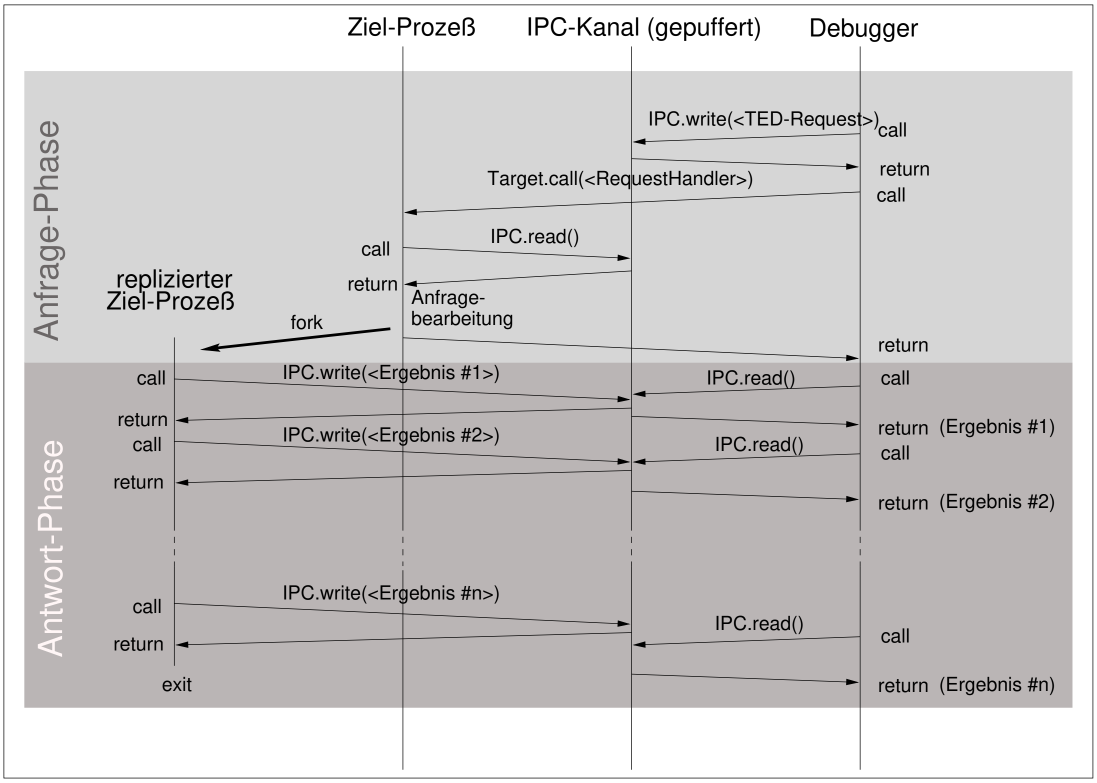

Abbildung 6.9.: Kommunikation mit repliziertem Ziel-Prozeß

beteiligten Partner. In der Anfragephase schreibt der Debugger die Anfragedaten in den IPC-Kanal und ruft die Funktion im Ziel-Prozeß. Im Ziel-Prozeß werden die Ergebnisdaten berechnet und der Prozeß wird dupliziert. In der Antwortphase kommuniziert der Debugger ausschließlich mit dem replizierten Ziel-Prozeß. Die Synchronisation erfolgt dabei implizit über blockierende Lese- bzw. Schreiboperationen[7] auf dem IPC-Kanal. Der Vorteil dieser Lösung ist, daß nur einmal die Signalisierung des Zielprozesses notwendig ist und demzufolge die Gefahr eines geänderten Verhaltens der Zielapplikation geringer ist.

[7] Eine Schreiboperation blockiert solange der Puffer voll ist.

Eine weitere Alternative, die häufige wechselseitige Ausführung von Debugger und Applikation zu vermeiden, besteht darin, die Empfangsroutinen im Debugger nebenläufig zu den Kontrollfunktionen für den Ziel-Prozeß auszuführen. Diese Möglichkeit wurde nicht in Betracht gezogen, da die Integration von Nebenläufigkeit und damit die Benutzung einer Threadbibliothek eine schwere Designentscheidung für die jeweilige Debugger-Implementierung bedeutete. Der Fokus dieser Arbeit liegt auf der Bereitstellung einer Schnittstelle, die von herkömmlichen Debugger-Lösungen benutzt werden kann. Der im Abschnitt 4.2 vorgestellte *KDB* besteht aus mehreren Threads, sodaß eine einfachere Form der Kommunikation zwischen Debugger und Applikation möglich ist.

6.4.3. Benutzte IPC-Varianten

Für die nachrichtenorientierte Datenübertragung stehen die IPC-Varianten Pipe oder FIFO, Message-Queue sowie die Varianten der Netzwerkkommunikation (Socket, TLI usw.) zur Verfügung. Generell erscheint es möglich, jede der Formen zu nutzen, obwohl sich beispielsweise die Etablierung einer TCP-Verbindung als schwierig erweist, da beiden Kommunikationspartnern während des Verbindungsaufbaus wechselseitig die Ausführung ermöglicht werden muß. Hier könnte jedoch eine Art Server benutzt werden, der die Anfragen des Debuggers puffert. Der Ziel-Prozeß könnte nach der Signalisierung die Anfrage- und Ergebnisdaten mit dem Server austauschen. Im Anschluß daran würde der Debugger die Ergebnisdaten vom Server beziehen.

Da die Nutzung vieler IPC-Varianten, sowohl POSIX als auch System V, möglich erscheint, wurde dies bei der Implementierung des Kommunikationsmoduls des TDI berücksichtigt. Die derzeitige Lösung basiert auf der Nutzung gemeinsamen Speichers. Die Verwendung von generischen Kommunikationsroutinen gestattet es jedoch, beliebige IPC-Varianten oder Netzwerk-Transportprotokolle zu benutzen.

Benutzung von `ptrace` für den Datenaustausch

Der Datenaustausch könnte auch mit dem Systemruf `ptrace` erfolgen, indem der Debugger die Anfragedaten an eine definierte Stelle in das Prozeßabbild der Applikation schreibt. Mit einem `ptrace`-Ruf kann jedoch nur ein Speicherwort[8] gelesen und geschrieben werden, sodaß die Anzahl der Systemrufe proportional zur Datenmenge steigt. Die IPC-Variante Shared-Memory erfordert nur Systemrufe für das Anfordern und Freigeben des Speicherbereiches unabhängig von seiner Größe. Das Kopieren der Daten aus dem gemeinsamen Speicher erfolgt ohne Systemintervention.

[8] Die Größe eines Speicherwortes richtet sich nach der zugrundeliegenden Architektur. Auf den betrachteten 32-bit-Systemen ist die Wortgröße 4 Bytes.

Die Benutzung von `ptrace` für den Datenaustausch hätte den Vorteil, daß im TDI keine Kommunikationsroutinen implementiert werden müßten. Nachteilig sind jedoch die Ineffizienz und die Tatsache, daß die Pthreads-Implementierung unter Umständen mit erweiterten Symbolinformationen vorliegen muß, da ein ausgezeichnetes Symbol für den Datenzugriff nötig ist. Wird ein Datenbereich fester Größe in der Pthreads-Laufzeitumgebung festgelegt, über den der Datenaustausch erfolgt, so kann es wie bereits angedeutet zu Problemen bei großen Ergebnisdatenmengen kommen. Das Ergebnis müßte fragmentiert und wechselseitig übertragen werden.

Das folgende Diagramm (Abbildung 6.10) soll die Ineffizienz der Datenübertragung mit `ptrace` verdeutlichen. Exemplarisch wurde ein Debugger-Befehl implementiert, der die Priorität, die Adresse der Startfunktion und deren Argument sowie den implementierungsseitig kodierten Zustandswert aller Threads anzeigt. Es wurden die Antwortzeiten gemessen, die bei Benutzung des TDI (Shared

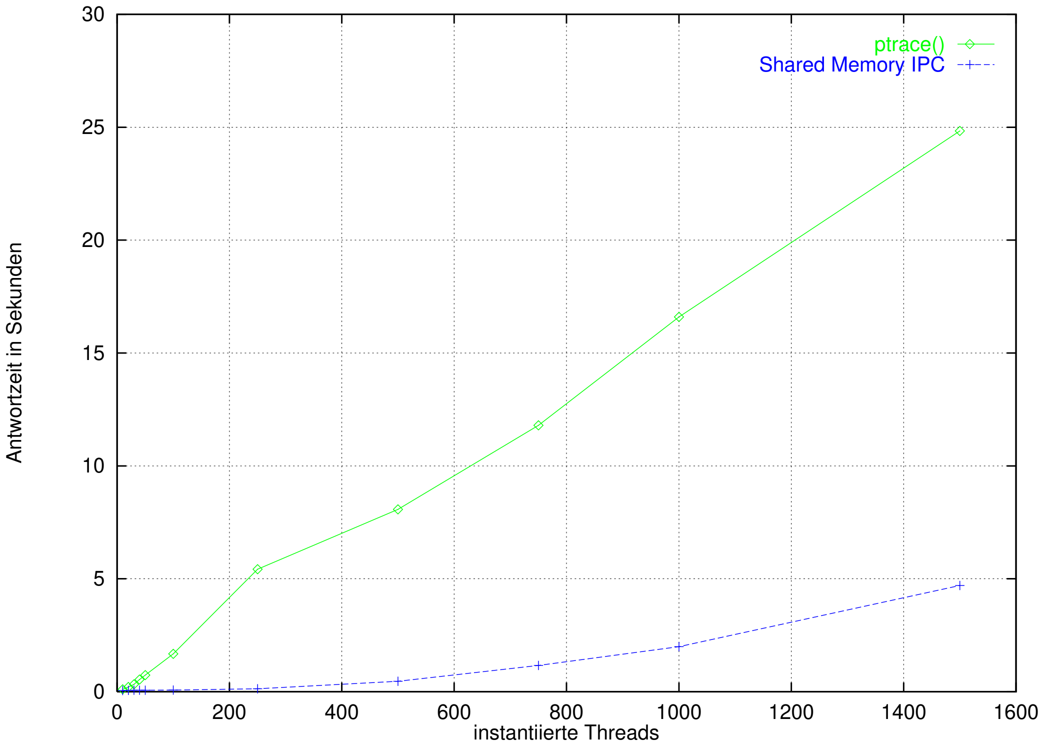

Abbildung 6.10.: Vergleich der Antwortzeiten: Shared Memory IPC und `ptrace`

Memory[9]) und der `ptrace`-Variante zu beobachten sind. Grundlage war die im Rahmen dieser Arbeit erweiterte FSU-Pthreads-Implementierung. Die `ptrace`-Variante entsprach der SmartGDB-Lösung [Hal96] für die FSU-Pthreads.

Es ist deutlich zu erkennen, daß die im TDI verwendete Methode des Datenaustausches zwischen Debugger und Zielapplikation effizienter ist.

[9]In diesem Fall wurde eine Seitengröße von 8 kByte benutzt.

6.4.4. Kernel-Threads

Im Gegensatz zur komplexen Lösung für User-Space-Threads kann die Kommunikation bei Kernel-Thread-Lösungen vereinfacht werden. Im folgenden werden Lösungsansätze für die Solaris Pthreads und die Linuxthreads angedeutet.

6.4.4.1. Solaris Threads

Bei der Solaris-Implementierung der POSIX-Threads werden LWP's (Lightweight Processes) verwendet. Da die Thread-Informationen vom System zur Verfügung gestellt werden, ist es möglich, diese Daten außerhalb des eigentlichen Applikationsprozesses zu lesen. Unter Solaris werden die Daten der Kernel-Threads (LWP's) im `/proc`-Dateisystem bereitgestellt. Die POSIX-Threads Objekte sind wie bei allen Pthreads-Implementierungen nur im User-Space sichtbar. Solaris stellt jedoch einen Mechanismus bereit, mit dem unter Zuhilfenahme der `/proc`-Dateien der LWP's und des Prozeß-Speicherabbildes die POSIX-Thread-Informationen sichtbar gemacht werden können. Hierzu verfügt Solaris über die Bibliothek `libthreaddb.so.1` [Suna], die Funktionen für das Auslesen der Pthreads-Informationen enthält. Nachteilig an dieser Lösung ist, daß keine Informationen über Mutexe und Bedingungsvariablen verfügbar sind. Hier liegt es jedoch bei Sun, eine Unterstützung zu realisieren. Zusammenfassend ist es möglich, daß ein vom Nutzer authorisierter Prozeß die POSIX-Threads-Daten des Zielprozesses ausliest und über die TED-Schnittstelle zur Verfügung stellt.

Der Datenaustausch könnte beispielsweise zwischen Debugger und einem `/proc`-Leseprozeß stattfinden. Die Abbildung 6.11 beschreibt die einzelnen Schritte der Kommunikation. Es ist zu erkennen, daß sich der TDI-Zugriff für den Debugger im Vergleich zur oben vorgestellten Lösung für die User-Space-Implementierungen nur unwesentlich ändert. Der einzige Unterschied besteht in der Signalisierung. Der `/proc`-Leseprozeß wird durch ein UNIX-Signal mittels `kill()` aktiviert und beginnt dann mit der Anfragebearbeitung. Daß die Kommunikation mit einem eigenständigen TDI-Prozeß und nicht mit dem Zielprozeß erfolgt, ist für den Debugger unerheblich.

6.4.4.2. Linuxthreads

Wird bei User-Level-Implementierungen der Prozeß angehalten, so werden auch alle in ihm vorhandenen Threads gestoppt. Unter Linux werden Threads jedoch als Prozesse verwaltet. Um alle Threads zu stoppen, muß jedem zugehörigen Prozeß ein Stopp-Signal gesendet werden. Insbesondere ist es unter Linux möglich, nur eine Teilmenge der Threads zu stoppen. Das eröffnet die Perspektive für eine weitere Möglichkeit des Datenaustausches.

Man instantiiert bei der Initialisierung der POSIX-Threads-Laufzeitumgebung einen sogenannten TDI-Thread, der dem Debugger bekannt sein muß. Wird die

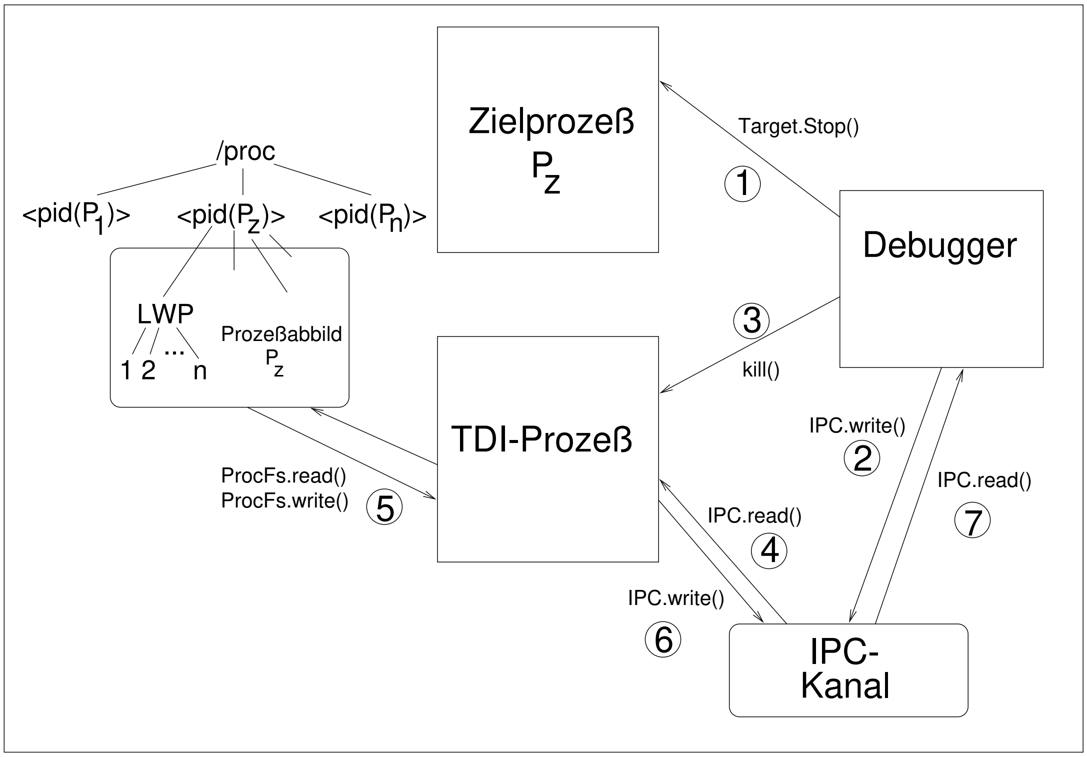

Abbildung 6.11.: *Solaris Pthreads*: Kommunikation mit eigenständigem TDI-Prozeß

Applikation vom Debugger gestoppt, so bedeutet daß, das alle Prozesse, die mit der Applikation assoziiert sind, angehalten werden müssen. Wird der TDI-Thread nicht gestoppt, so kann er ähnlich dem oben erwähnten `/proc`-Leseprozeß die Daten in gleicher Weise mit dem Debugger austauschen. Die Signalisierung erfolgt ebenfalls mit UNIX-Signalen. Die Abbildung 6.12 verdeutlicht diese Überlegung.

6.4.4.3. Auswirkungen auf die geforderte Einheitlichkeit der TDI-Benutzung

Die beschriebenen Möglichkeiten zur Kommunikation zwischen TDI und Debugger bei Pthreads-Implementierungen auf der Grundlage von Kernel-Threads wirkt sich nicht auf den Gebrauch der IPC-Primitiven und damit auf den TDI-Zugang für den Debugger aus. Lediglich die Signalisierung erfolgt anders als bei den User-Space-Implementierungen mit UNIX-Signalen. Ein aufwendiges Rufen der `RequestHandler`-Funktion des TDI entfällt damit. Aufgrund der bestehenden Signalisierungsalternativen erwächst für den Debugger die Notwendigkeit, den Typ der Pthread-Implementierung (Kernel oder User-Space) zu kennen, um die Signalisierung entsprechend vorzunehmen. Die Erkennung des Implementierungstyps ist

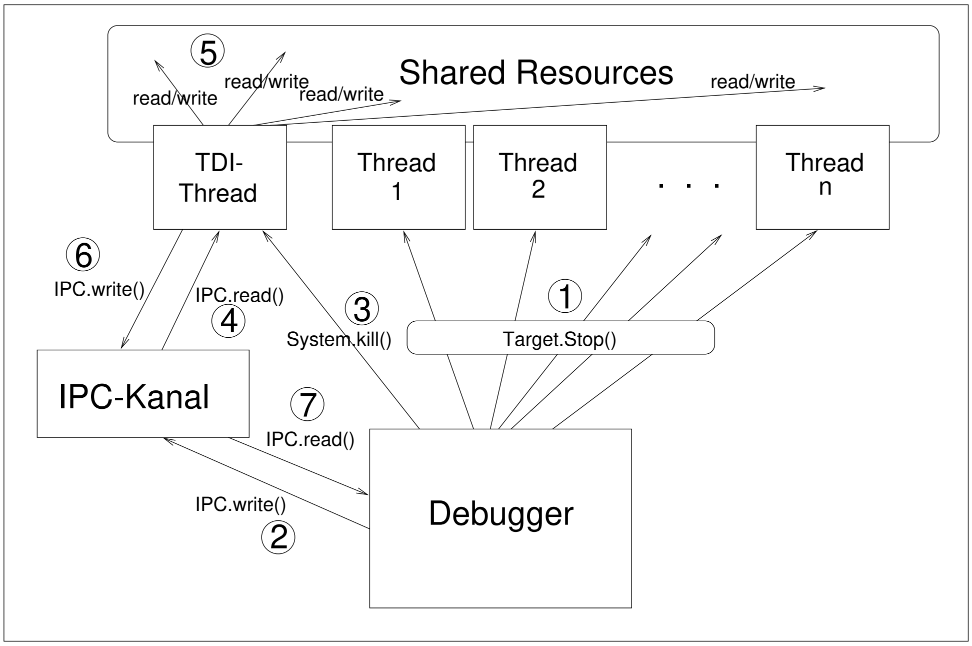

Abbildung 6.12.: *Linuxthreads:* Kommunikation mit TDI-Thread

in Abschnitt 6.6.1 beschrieben.

Die unterschiedliche Signalisierung macht es notwendig, daß der Debugger Informationen über die Klasse der jeweiligen Implementierung erhält. Bei reinen User-Space-Implementierungen muß er eine Funktion im Zielprozeß rufen. Bei Kernel-Thread-Implementierungen und Mischformen sendet er dem TDI-Thread ein definiertes UNIX-Signal, wobei im Vorfeld die Prozeß- oder Thread-ID des TDI-Threads an den Debugger übergeben werden muß.

Da die Anzahl der Pthreads-Klassen bekannt und fest ist, ist eine Erweiterung der Signalisierungsfunktionalität des Debuggers bei Unterstützung aller Pthreads-Varianten (Kernel-Threads,User-Level-Threads und Mischformen) ausgeschlossen.

6.5. Verwaltungs- und Hilfsfunktionen des TDI

Eine weitere Komponente des TDI stellen die Verwaltungs- und Hilfsfunktionen dar. Sie kompensieren die fehlende Funktionalität der POSIX-Threads-Implementierungen für ein Thread-Debugging. Die folgenden Abschnitte weisen auf die in diesem Zusammenhang erforderliche Funktionalität hin und diskutieren Lösungsansätze zu deren Realisierung.

6.5.1. Registrierung von TED-Funktionen

Die Attribute der in Abschnitt 6.2.1 definierten Objekte der TED können mit Hilfe der im TED-Modul bereitgestellten Funktionen gelesen und geschrieben werden. Weitere Funktionen ermöglichen die Mengeniteration über die instantiierten Objektklassen. Die Menge der TED-Funktionen sind der Befehlssatz, auf dem die generische Anfragebearbeitung des TDI basiert. Da jede Pthreads-Implementierung unterschiedliche Funktionen zur Unterstützung dieses Befehlssatzes bereitstellt, müssen diese dem TDI zur Laufzeit bekannt gemacht werden. Die Bekanntgabe erfolgt zur Laufzeit, da das TDI dynamisch geladen werden kann.

Eine Möglichkeit wäre es, für die Attributzugriffsfunktionen einer Objektklasse einen Funktionscontainer zu definieren, der alle Funktionen enthält. Diese Container könnten dann vom TED für jede einzelne Klasse gefüllt und der TDI-Schnittstelle übergeben werden. Im Hinblick auf die Erweiterung der TED-Funktionalität ist dieser Ansatz ungeeignet. Die Erweiterung einer Relation im TED um ein Attribut hätte die Änderung des entsprechenden Funktionscontainertyps zur Folge. Daraufhin müßten alle TED- und somit auch die jeweiligen Pthreads-Implementierungen angepaßt, das heißt neu erzeugt werden.

Das TDI stellt eine flexible Variante zur Verfügung, die vom TED bereitgestellte Funktionalität zu registrieren, um die gefordete Erweiterbarkeit des TDI ohne Konsequenzen für die Pthreads-Implementierungen zu ermöglichen. Vom TED müssen nur die Funktionen beim TDI registriert werden, die auch von ihm bereitgestellt werden. Dazu werden vom TDI folgende Registrierungsfunktionen zur Verfügung gestellt:

Registrieren der Iteratoren :

```
int SetIterFunc (RelT Rel, ObjRefT (*GetFirst) (),
                           ObjRefT (*GetNext)  (ObjRefT Last)
```

Registrieren der Attributzugriffsfunktionen :

```
int SetAttrFunc (RelT Rel, AttrT Attr,
                 AttrDomainT (*GetFunc) (ObjRefT Obj),
                 AttrDomainT (*SetFunc) (ObjRefT Obj,
                                         AttrDomainT value))
```

Mit `SetIterFunc` können zu einer im TED definierten Relation (bisher Thread, Mutex und Bedingungsvariable) die Iteratorfunktionen `GetFirst` und `GetNext` registriert werden. `SetAttrFunc` erlaubt es, dem Attribut `Attr` der Relation `Rel` die Lese- und die Schreibfunktion zuzuordnen.

Für die Zukunft ist es nicht auszuschließen, daß noch weitere Objekte, Attribute oder Beziehungen im TDI definiert werden. In diesem Falle wären TDI-Änderungen bezüglich der TDI-QL-Grammatik notwendig, die zu einer neuen TDI-Version führten. Für ältere oder unvollständige TED-Implementierungen hätte das keine Änderungen zur Folge, da die Registrierungsfunktionen des TDI unverändert blieben.

6.5.2. Bereitstellung von Objektcontainern

Das POSIX-Threads-API stellt keine Möglichkeiten zur Verfügung, über die Menge der im Standard definierten und zur Laufzeit instantiierten Objekte zu iterieren. Die Funktionen des Pthreads-API stellen lediglich die Möglichkeit zur Verfügung, Objekte zu erzeugen, zu manipulieren und zu verwerfen. Die Objekte sind der Pthreads-Implementierung nur zum Zeitpunkt der Ausführung der API-Funktionen als Parameter bekannt. Eine Ausnahme bilden die Thread-Objekte, da sie dem Scheduler und für die Stack-Verwaltung bekannt sein müssen. Für Mutexe und Bedingungsvariablen besteht das Problem der fehlenden Registrierung.

Für die Registrierung der instantiierten Objekte wird hier der folgende Mechanismus vorgeschlagen. Das TDI stellt für jeden Objekttyp die folgenden Registrierungsfunktionen bereit:

Einfügen eines Objektes :

```
int RegisterObject   (RelT Rel, ObjRefT ObjRef)
```

Entfernen eines Objektes :

```
int UnregisterObject (RelT Rel, ObjRefT ObjRef)
```

Test auf Enthaltensein :

```
int     IsRegistered (RelT Rel, ObjRefT ObjRef)
```

Die Registrierungsfunktionen sollten von der Pthreads-Implementierung bei jeder Instantiierung beziehungsweise beim Entfernen eines Objektes verwendet werden. Hierbei entsteht ein neues Problem. Die Funktionen der Pthreads Implementierung müssen zur Laufzeit Kenntnis davon haben, ob die Thread-Debug-Schnittstelle involviert ist, da nur dann eine Registrierung erforderlich und möglich ist. Dazu wird hier die folgende Methode angewendet. Der Pthreads-Implementierung wird eine globale Variable (zum Beispiel `pthread_debug_with_TDI`) hinzugefügt, die bei normaler Ausführung der Thread Applikation den Wert `0` besitzt. Ist die Applikation dem Debugging unterzogen, so wird mit Hilfe des Debuggers die Variable `pthread_debug_with_TDI` mit einem Wert ungleich `0` beschrieben. Die Thread-Laufzeitumgebung registriert sich während ihrer Initialisierung abhänging von `pthread_debug_with_TDI` beim TDI und erhält im Gegenzug Referenzen auf die Objektregistrierungsfunktionen, die es fortan benutzen kann.

Werden die Objektregistrierungsfunktionen vom TED verwendet, so erfolgt die Verwaltung der Objektmengen im TDI. Die Implementierung der Iteratorfunktionen im TED kann jetzt ebenfalls mit Hilfe des TDI geschehen. Dafür werden die folgenden Funktionen vom TDI angeboten:

Zugriff auf erstes Element :

```
AddrT GetFirstObject  (RelT Rel)
```

Zugriff auf Folgeelement :

```
AddrT GetNextObject   (RelT Rel, AddrT LastObjAddr)
```

Mit Hilfe dieser Funktionen kann über die mit `RegisterObject` und `Unregister-Object` erzeugte Menge von Objekten iteriert werden.

Die Benutzung der hier vorgestellten Funktionen stellt nur ein Angebot für die Entwickler der POSIX-Threads-Implementierungen dar, die die TED-Erweiterung realisieren wollen. Da die Registrierfunktionen unabhängig von der jeweiligen Pthreads-Implementierung sind, ist die Bereitstellung im TDI konsequent. Die Objektregistrierung kann jedoch auch proprietär im TED und damit in der Pthreads-Implementierung erfolgen.

6.5.3. Persistente ID's

Im POSIX-Threads-API ist der Thread-Typ mit `pthread_t` definiert. `pthread_t` ist meist als Zeigertyp definiert und der Wert einer Variablen vom Typ `pthread_t` ist die Adresse einer Thread-Struktur. Gegeben sind zwei Threads A und B vom Typ `pthread_t`. Angenommen, Thread A wird erzeugt, führt seine Funktion aus und wird dann beendet. Im Anschluß wird Thread B erzeugt. Es ist nun möglich, daß $B = A$ ist, da die Struktur für den Thread B bei der Speicheradresse der Struktur von Thread A angelegt wurde. Im POSIX-Standard werden Threads über ihre `pthread_t`-Werte identifiziert, wie die Definition der Funktion `pthread_equal` zeigt. Die ID eines POSIX-Threads ist damit nur für die Dauer seiner Existenz persistent. Sie kann danach von einem anderen Thread angenommen werden.

Für das Debugging ist es notwendig, daß Thread-ID's prozeßpersistent sind, das heißt für die Dauer des Prozesses gültig sind. Bei Applikationen mit einer hohen Thread-Fluktuation wäre es sonst nicht mehr bestimmbar, welche Threads zu einem bestimmten Zeitpunkt instantiiert sind.

Die Bereitstellung der in Abschnitt 6.5.2 beschriebenen Registrierungsfunktionen für die Pthreads-Objekte ist eng mit der Lösung des Persistenzproblems verbunden. Bei jeder Registrierung eines Objektes wird eine globale Variable inkrementiert und als ID zurückgegeben. Kommt es während der Ausführung des Prozesses nicht zu einem Überlauf dieses Wertes, so sind alle vergebenen ID's prozeßpersistent.

Für TED-Implementierungen gilt damit die Empfehlung, die Registrierfunktionen der Objektcontainer auch für Threads zu benutzen, um persistente Thread-ID's zu gewährleisten.

6.6. TDI-Perspektiven

Nachdem die einzelnen Komponenten des TDI, genauer des Schnittstellenkerns, betrachtet wurden, sollen die sich ergebenden Perspektiven für die Anrainer der Schnittstelle beschrieben werden. Die Anrainer der Schnittstelle sind zum einen der Debugger als Nutzer der Schnittstelle und auf der anderen Seite die Pthreads-Implementierung als Erbringer der für die Anfragebearbeitung notwendigen Basisfunktionalität.

6.6.1. Debugger als TDI-Klient

Zur Laufzeit bilden der TDI-Kern und die jeweilige Pthreads Laufzeitumgebung eine Einheit, die als TDI-Server betrachtet werden kann. Der Debugger als Anfragesteller und Nutzer der Schnittstelle hat die Rolle des TDI-Klienten. Die Kommunikation zwischen Server und Klient erfolgt wie in Abschnitt 6.4 dargesetllt über eine beliebige Form der Interprozeßkommunikation (IPC).

Die Funktionalität für die Ausführung des Debuggers als TDI-Klient kann in einem Modul zusammengefaßt und als Bestandteil des TDI-Softwarepaketes zur Verfügung gestellt werden. Es entspricht der Philosophie der Thread-Debug-Schnittstelle, daß die von Debugger und Pthreads unabhängige Funktionalität von ihr erbracht wird. Da komplexe Operationen im Klientenmodul von einfach zu benutzenden Funktionen gekapselt sind, kann die in Abschnitt 5.3 gestellte Forderung nach einem möglichst einfachen Schnittstellenzugang erfüllt werden. Der TDI-Zugang erfolgt damit nicht direkt über den IPC-Kanal, sondern mit Hilfe der Klient-Funktionen.

Die Abbildung 6.13 beschreibt die Verteilung der TDI-Komponenten zur Laufzeit. Das in der POSIX-Threads-Laufzeitumgebung enthaltene TED-Modul bildet zusammen mit den TDI-Komponenten den TDI-Server, der im Applikations-Prozeß enthalten ist. Der Debugger kommuniziert mit Hilfe des TDI-Klient-Moduls mit dem Server.

Diese Vereinfachung des Schnittstellenzuganges ist jedoch mit einem Mehraufwand bei der TDI-Implementierung verbunden, da die Sprachunabhängigkeit des IPC-Zuganges durch die Benutzung der quellsprachgebundenen Klient-Funktionen

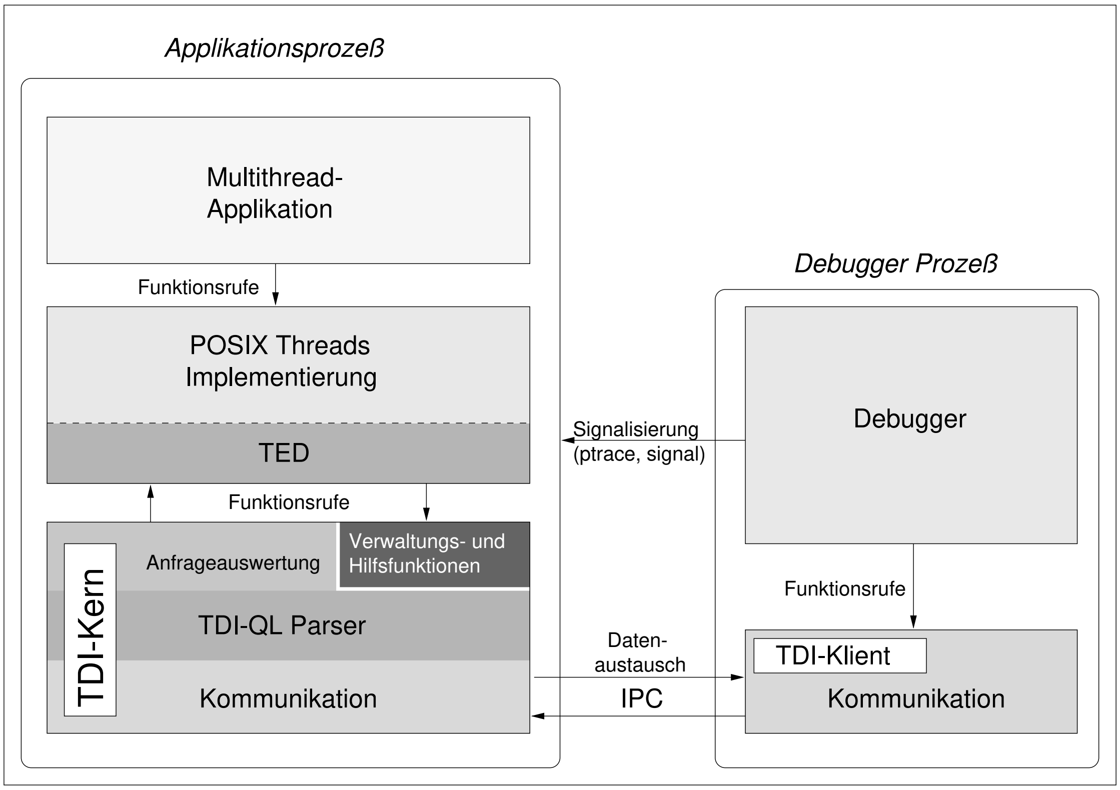

Abbildung 6.13.: TDI-Verteilung zur Laufzeit

ersetzt wird. Für jede Debugger-Quellsprache muß demzufolge ein TDI-Klient bereitgestellt werden[10].

Die Benutzung des Klient-Moduls gestaltet sich sehr einfach. Es wird eine Instanz des abstrakten Datentyps `Klient` erzeugt, der die folgenden Methoden anbietet:

Öffnen des Klienten :

```
int open ()
```

Senden einer TDI-QL Anfrage :

```
int send (char* AnfrageText, int AnfrageLaenge)
```

Empfangen des Anfrage-Ergebnisses :

```
int receive (char** ErgebnisText, int* ErgebnisLaenge)
```

[10] Da der GNU Debugger (GDB) exemplarisch für die Nutzung der TDI-Schnittstelle angepaßt wurde, steht die Klient-Komponente derzeit nur in der Programmiersprache C bereit.

Schließen des Klient-Objektes :

```
int close ()
```

Die Methoden haben folgende Semantik. Mit `open` werden interne mit dem Klient-Objekt assoziierte Datenstrukturen angelegt und initialisiert. Unter anderem wird auch der IPC-Kanal konfiguriert. Diese Konfiguration erfolgt zur Laufzeit durch Benutzereingabe oder durch das Einlesen von Konfigurationsdateien, um die Übersetzung des Debuggers für eine IPC-Neukonfiguration zu vermeiden.

Mit den Methoden `send` und `receive` erfolgt der Datenaustausch mit dem TDI-Kern über den konfigurierten IPC-Kanal, wobei es eine Besonderheit zu beachten gilt. Wie in Abschnitt 6.4 erläutert, muß nach dem Senden der Anfragedaten eine Signalisierung der gestoppten Applikation erfolgen. Die Signalisierung ist Aufgabe des Debuggers und demzufolge für jeden Debugger unterschiedlich gelöst. Sie kann deshalb nicht Aufgabe des TDI's sein. Nach dem Rufen der Methode `send` muß eine im Debugger implementierte Funktion zur Notifizierung der Applikation gerufen werden.

Um bei User-Space-Implementierungen eine Signalisierung der Applikation zu ermöglichen, muß dem Debugger ein ausgezeichnetes Symbol für die in Abschnitt 6.4.2 erwähnte `RequestHandler`-Funktion bekannt sein. Er muß die Code-Adresse der Funktion kennen, um ihren Ruf vorzubereiten. Das TDI stellt folgende `RequestHandler`-Funktion bereit:

TDI-Signalisierungsfunktion :

```
int TDI_Server (char* IPCSpec)
```

Mit dem Parameter `IPCSpec` wird der Typ der vom TDI-Klienten geöffneten IPC-Verbindung angegeben. Die Konfigurierbarkeit ist aus Portabilitätsgründen notwendig. Um bei der Wahl des Parametertyps eine Einschränkung der Kodierungsmöglichkeiten zu vermeiden, wurde eine frei interpretierbare Zeichenkette gewählt. Der Debugger muß in der Lage sein, diesen Parameterstring in das Speicherabbild des Ziel-Prozesses zu kopieren und die Funktion `TDI_Server` mit diesem Argument aufzurufen.

Die Zeichenkette `IPCSpec` besteht aus den zwei Teilen <`IPC-Variante`> und <`Spezifische Parameter`>. Das Argument für den Parameter <`IPC-Variante`> muß einer standardisierten Menge von Bezeichnern entstammen. Da in der vorliegenden Version des TDI nur die IPC-Variante Shared Memory implementiert wurde, enthält diese Menge derzeit nur das Element "`SV_IPC_SM`". Das Argument für den Parameter <`Spezifische Parameter`> ist eine durch Leerzeichen getrennte Liste von Elementen der Form <`key`>=<`value`>. Die Menge der Schlüsselwörter ist dabei ohne die Änderung des Prototypen von `TDI_Server` erweiterbar. Im Falle der Benutzung von Shared Memory wären beispielsweise Schlüsselwörter für die Parameter Shared-Memory-ID und die Segmentgröße denkbar.

Die Bereitstellung eines ausgezeichneten Funktionssymbols zur Signalisierung ist nur notwendig, wenn es sich um eine User-Space-Implementierung der Pthreads handelt. Bei Kernel-Thread-Implementierungen erfolgt die Signalisierung wie im Abschnitt 6.4.4 erwähnt mit UNIX-Signalen. In diesem Fall kann die Konfiguration für den IPC-Kanal nicht als Signalisierungs-Parameter übergeben werden. Die Zeichenkette für die Konfiguration muß in den Ziel-Prozeß geschrieben werden. Dafür wird ein ausgezeichnetes Symbol vom Typ `char*` benötigt:

TDI-Zeichenkette für die IPC-Konfiguration (Kernel-Threads) :

```
char* TdiIPCSpec
```

Nach der Signalisierung wird die Zeichenkette `TdiIPCSpec` im TDI-Server ausgelesen, um den IPC-Kanal in geeigneter Weise zu öffnen.

Da es die beschriebenen Alternativen der Signalisierung gibt, muß dem Debugger bekannt sein, um welche Art von Pthread-Implementierung es sich handelt. Die Lösung dieses Problems wurde mit der des Problems der TDI-Aktivierung verbunden. Forderung an das TDI ist seine Optionalität. Das TDI sollte deshalb nur mit der Multithread-Applikation gebunden werden, wenn ein Debugging erwünscht ist. Um diesen Wunsch erkennbar zu machen, wird in der Pthreads-Implementierung ein Symbol definiert, das vom Debugger gelesen und geschrieben werden kann:

Indikator-Variable der Pthread-Implementierung :

```
int pthread_debug_with_TDI
```

Die Variable `pthread_debug_with_TDI` erfüllt zwei Aufgaben. Zum einen wird sie in der Pthreads-Implementierung auf einen Wert gesetzt, der den Typ der Implementierung erkennen läßt. Der Debugger kann den Implementierungstyp lesen, um die Signalisierungsart entsprechend zu wählen. Weiterhin wird der Pthreads-Implementierung durch Setzen der Variable auf einen definierten Wert signalisiert, daß sie das TDI binden kann und dem Debugging unterzogen wird.

6.6.2. POSIX-Thread-Implementierung

Wie Abbildung 6.13 zeigt, hat der TDI-Kern ausschließlich Zugriff auf die TED-Komponente der POSIX-Threads-Implementierung. Das POSIX-Threads-API wird nicht zur Erbringung der Debug-Funktionalität benutzt. Um die TDI-Schnittstellenperspektive der POSIX-Threads-Implementierung zu betrachten, ist es ausreichend, die Sicht des TED auf die Thread-Debug-Schnittstelle zu analysieren.

Die TED-Implementierung muß gewisse Richtlinien erfüllen, um vom TDI benutzt werden zu können. Eine Forderung an die TDI-Schnittstelle war die Optionalität der Benutzung. Es ist demzufolge nicht möglich, Funktionen des TDI im TED direkt zu rufen, da das TDI-Modul bei nicht benötigter Debug-Unterstützung auch nicht gebunden wird. Für das TED-Modul muß jedoch die Möglichkeit bestehen, die vom TDI angebotene Funktonalität zu nutzen. Nur so können die vom TED bereitgestellten Funktionen registriert werden und die Benutzung des TDI-Service zur Bereitstellung von Mengencontainern und der Vergabe von prozeßpersistenten ID's wird möglich.

Für die Auflösung des beschriebenen Konflikts wurde folgende Lösung erarbeitet, die auf der indirekten Benutzung der TDI-Funktionen über Funktionszeiger basiert. Die Lösung sieht vor, daß dem TDI vom TED die Möglichkeit geboten wird, Funktionen zu registrieren. Im Gegensatz zur genannten Einschränkung für das TED bezüglich der direkten Funktionsaufrufe können im TDI Funktionen des TED direkt gerufen werden, da die Existenz des TDI die des TED-Moduls impliziert. Konkret muß im TED die folgende Funktion unter Verwendung des genannten Namens bereitgestellt werden:

TED-Funktion zur Registrierung von TDI-Funktionen :

```
void pthread_TDI_register(TDIAuxFuncsT* TDIAuxFuncs)
```

Die Funktion `pthread_TDI_register` wird vom TDI aufgerufen. Damit übergibt das TDI dem TED-Modul eine Liste von Funktionen, die später für die Registrierung der TED-Funktionen sowie für den Gebrauch der Verwaltungs- und Hilfsfunktionen des TDI benutzt werden können. Der Typ `TDIAuxFuncsT` muß der TED-Implementierung bekannt sein. Er ist wie folgt definiert:

```
typedef struct {

  /* Registrieren der TED Funktionen für: */
  /* i) den Attributzugriff               */

  int  (*SetAttrFunc) (RelT   Rel,
                       AttrT Attr,
                       AttrDomainT (*GetFunc) (ObjRefT),
                       AttrDomainT (*SetFunc) (ObjRefT, AttrDomainT));

  /* ii) die Mengeniteration              */

  int  (*SetIterFuncs) (RelT Rel,
                        ObjRefT (*getFirst) (),
                        ObjRefT (*getNext)  (ObjRefT));
```

```
  /* Funktionen für den Zugriff auf die Objektcontainer */

  int     (*RegisterObject)     (RelT Rel, ObjRefT ObjRef);

  int     (*UnregisterObject)   (RelT Rel, ObjRefT ObjRef);

  int     (*IsRegistered)       (RelT Rel, ObjRefT ObjRef);

  ObjRefT (*GetFirstObject)     (RelT Rel);

  ObjRefT (*GetNextObject)      (RelT Rel, ObjReftT Last);

} TDIAuxFuncsT;
```

`SetAttrFunc` und `SetIterFunc` sind die Referenzen auf die in Abschnitt 6.5.1 beschriebenen Hilfsfunktionen des TDI zum Registrieren der TED-Funktionalität. Die weiteren Elemente der Struktur sind Zeiger auf die in Abschnitt 6.5.2 beschriebenen TDI-Funktionen für die Verwaltung von Pthreads-Objekten in sogenannten Objektcontainern des TDI.

Die Festlegung auf den beschriebenen Typ `TDIAuxFuncsT` stellt keine Einschränkung für TDI-Weiterentwicklungen dar. Alle Funktionen sind unabhängig von der Anzahl der im TED definierten Relationen und ihrer Attribute, sodaß auch neu definierte Relationen und Attribute mit diesem Funktionssatz unterstützt werden können.

Nach der Bekanntgabe der benötigten Funktionsadressen durch das TDI werden diese Adressen den Funktionszeigern im TED zugewiesen, die korrespondierend zu den Funktionszeigern in der `TDIAuxFuncsT`-Struktur definiert sind. Bei normaler Verwendung der POSIX-Threads-Implementierung haben alle Funktionszeiger den Wert `NULL`, sodaß ein Dereferenzieren mit einem `NULL`-Test der Zeiger ausgeschlossen wird. Ist die Applikation jedoch dem Thread-Debugging unterworfen, so wurden bestimmte Funktionszeiger in der Funktion `pthread_TDI_register` gesetzt, sodaß die referenzierten TDI-Funktionen vom TED gerufen werden können.

Zusammenfassung

Die mit `pthread_TDI_register` bekanntgegebenen Funktionen sind der für die POSIX-Threads-Implementierung sichtbare Teil des TDI. Die Beschränkung auf einen kleinen Satz von übergebenen Funktionen ermöglicht eine weitestgehend unabhängige Entwicklung von TDI und TED, das heißt nicht jede Änderung des TDI zieht eine Anpassung des TED nach sich. Insbesondere soll hiermit eine längstmögliche Debug-Unterstützung für Pthreads-Implementierungen erreicht werden, die nicht mehr weiterentwickelt werden, jedoch die TED-Funktionalität aufweisen.

7. Implementierung

Nachdem das letzte Kapitel die Architektur des *Thread Debug Interface* beschrieben hat, sollen einige Implementierungsdetails näher betrachtet werden. Die Realisierung des TDI-Entwurfs ist mit einer Vielzahl von Implementierungsaufgaben in den unterschiedlichsten Bereichen verknüpft. Zum einen muß der generische Kern der Thread-Debug-Schnittstelle geschaffen werden. Daraufhin müssen bestehende Pthreads-Implementierungen modifiziert und um die TED-Komponente ergänzt werden, um ein Thread Debugging im Verbund mit dem TDI zu ermöglichen. Nach der Bereitstellung dieser Basisfunktionalität ist es möglich, einen Debugger an das TDI anzupassen.

Die folgenden Abschnitte sollen Details der Implementierung beleuchten, die bei vergleichbaren Implementierungen beachtet werden müssen. Ziel ist es insbesondere, die Gemeinde der Entwickler zu befähigen, andere POSIX-Threads-Implementierungen im Hinblick auf das TDI zu modifizieren.

7.1. TDI-Kern

Bei der Implentierung des TDI fiel besonders eine Problematik im Zusammenhang mit der Einbindung der TDI-Komponente in die Laufzeitumgebung der jeweiligen Pthreads-Implementierung auf. Die Ausführung der Funktion `TDI_Server` kann unter gewissen Umständen eine ungewünschte Ausführung von Teilen der Ziel-Applikation und damit ein anormales Applikationsverhalten hervorrufen. Es muß gewährleistet werden, daß die Funktionen des TDI-Kerns möglichst isoliert, das heißt ohne Wechselwirkungen zum Pthreads-Laufzeitsystem ausgeführt werden.

Neben dem Experimentieraufwand, der mit der Einbettung des TDI-Kerns in die Pthreads verbunden war, lag ein großer Teil der Arbeit in der Realisierung der Anfragebearbeitung. Diesem Realisierungsabschnitt ist deshalb ein weiterer Abschnitt gewidmet.

7.1.1. Probleme der TDI-Einbindung

Die für den Datenaustausch und die Anfragebearbeitung erforderlichen Funktionen des TDI-Kerns müssen von einem Thread in der Zielapplikation ausgeführt werden. Für Kernel-Thread-Implementierungen kann das ein separater Thread sein, bei User-Space-Implementierungen wird der TDI-Kern im Kontext des aktuellen Threads ausgeführt. Es entstehen Probleme, wenn der den TDI-Kern ausführende Thread blockiert oder von einem anderen Thread unterbrochen wird.

Die angesprochene Problematik läßt sich nach der Ursache des Schedulings unterteilen in:

i) die Ausführung der Funktion `TDI_Server` und der damit verbundenen Zuteilung von CPU-Zeit sowie

ii) der Auslösung des Schedulings durch den Ruf überladener Standard-Funktionen.

Der in **i)** angeführte Sachverhalt tritt auf, wenn Timer zur Gewährleistung eines zeitscheibenbasierten Schedulings (*Round Robin*) benutzt werden oder generell unter Verwendung von Signalen programmiert wird.

Bei der Ausführung des TDI-Kerns im Zuge der Anfragebearbeitung erhält der Zielprozeß CPU-Zeit und ist somit bereit, Signale zu bearbeiten. Timer können ablaufen, deren Fortschritt mit der Ausführung des Prozesses verknüpft ist - sogenannte virtuelle Timer. Für Echtzeit-Timer, die unabhängig von der Prozeßausführung ablaufen, kann bereits zum Zeitpunkt des Rufes der TDI-Funktion ein Signal vorliegen. Weitere Ursache für das Eintreffen von Signalen ist deren Generierung in anderen Prozesen (`kill()`-Systemruf). Für die Ausführung der TDI-Funktionen im Zielprozeß muß sichergestellt sein, daß keine Signalbehandlung erfolgt. Eine Signalbehandlung könnte die Ausführung der TDI-Routinen unterbrechen und möglicherweise andere Threads aktivieren.

Eine andere in **ii)** dargestellte Gefahr des Schedulings liegt in der Benutzung von überladenen Standard-C-Funktionen durch den TDI-Kern, die ein Scheduling auslösen können. Viele Pthreads-Pakete enthalten eine ganze Reihe von überladenen Standard-Funktionen. Grund dafür sind die fehlende Reentranz beziehungsweise der prozeßblockierende Charakter einiger Ein- und Ausgabefunktionen.

Da das TDI mit der Applikation gebunden wird, nutzt es auch deren Bibliotheken. Es ist aufgrund des globalen Namensraumes für C-Funktionen nicht möglich, die Verwendung gleichnamiger, jedoch anders implementierter Funktionen im TDI-Modul zu wählen. Werden im TDI die überladenen Funktionen aus der Pthreads-Bibliothek gerufen, so können diese Rufe bei der Erlangung von sogenannten Funktions-Locks blockieren und ein anderer Thread der Applikation bekäme CPU-Zeit. Ein Auslösen des Schedulings während des Aufrufes der `TDI_Server`-Funktion hätte zur Folge, daß andere Threads während des Auslesens der TED-Relationen ausgeführt würden, was die Qualität des Anfrage-Ergebnisses in Frage stellte. Eine Benutzung überladener Funktionen im TDI muß daher vermieden werden.

Im folgenden sollen einige Maßnahmen zur Prävention ungewollten Schedulings beschrieben werden.

7.1.1.1. Behandlung von Signalen

Um ein durch das Eintreffen von Signalen verursachtes Scheduling zu vermeiden, wurden im Laufe der Entwickling des TDI-Kerns zwei Möglichkeiten betrachtet. Im ersten Ansatz wird versucht, das Eintreffen von Signalen *während* der TDI-Ausführung zu verhindern. Da dies nicht garantiert werden kann, wurde ein zweiter, aufwendigerer Ansatz verfolgt. Grundlegende Idee dabei war, nicht das Eintreffen der Signale zu verhindern, sondern deren Behandlung (Scheduling) für die Zeit der TDI-Ausführung zu deaktivieren.

Maskieren und Demaskieren von Signalen

Um das Auslösen von Signalbehandlungsroutinen zu verhindern, könnten die Signale vor der TDI-Ausführung maskiert sowie danach demaskiert werden. Da Maskieren und Demaskieren der Signale keine atomaren Operationen sind, ist das Eintreffen von Signalen während der Ausführung dieser Operationen nicht auszuschliessen. Besonders bei Echtzeit-Timern, die während der Stopp-Phase des Zielprozesses abgelaufen sind und deren Signale beim Ruf des TDI's bereits vorliegen, besteht die Gefahr, daß die Signalbehandlungsroutinen vor dem Maskieren ausgeführt werden. Es wäre in diesem Zusammenhang wünschenswert, die Maskierung von Signalen prozeßextern vorzunehmen. Es gibt jedoch keine UNIX-Standardbefehle zum Maskieren der eintreffenden Signale eines anderen Prozesses.

Deaktivieren der Signalbehandlung

Eine andere Möglichkeit besteht darin, nicht das Eintreffen von Signalen zu verhindern, sondern deren Behandlung zu beeinflussen. Die entsprechende Lösung wurde nicht im TED, sondern in der POSIX-Threads-Implementierung realisiert, da hier die Signalbehandlung und ein mögliches Scheduling erfolgt. Abbildung 7.1 skizziert die im folgenden beschriebene Vorgehensweise. Die Pthreads-Implementierung muß einen Schalter (Flag) anbieten, der von außen durch den Debugger gesetzt werden kann und eine Behandlung respektive Nichtbehandlung von Signalen bewirkt. Dieses Flag wird von der Signalbehandlungsroutine der POSIX-Threads-Implementierung gelesen. Ist die Behandlung von Signalen erlaubt, so kann bei Eintreffen der Timer-Signale ein Scheduling erfolgen. Andernfalls kehrt die Signalbehandlungsroutine zurück.

Einmal ignorierte Signale müssen jedoch nach Ausführen des TDI-Kerns wieder an den Zielprozeß gesendet werden. Dies ist unter anderem notwendig, um eine

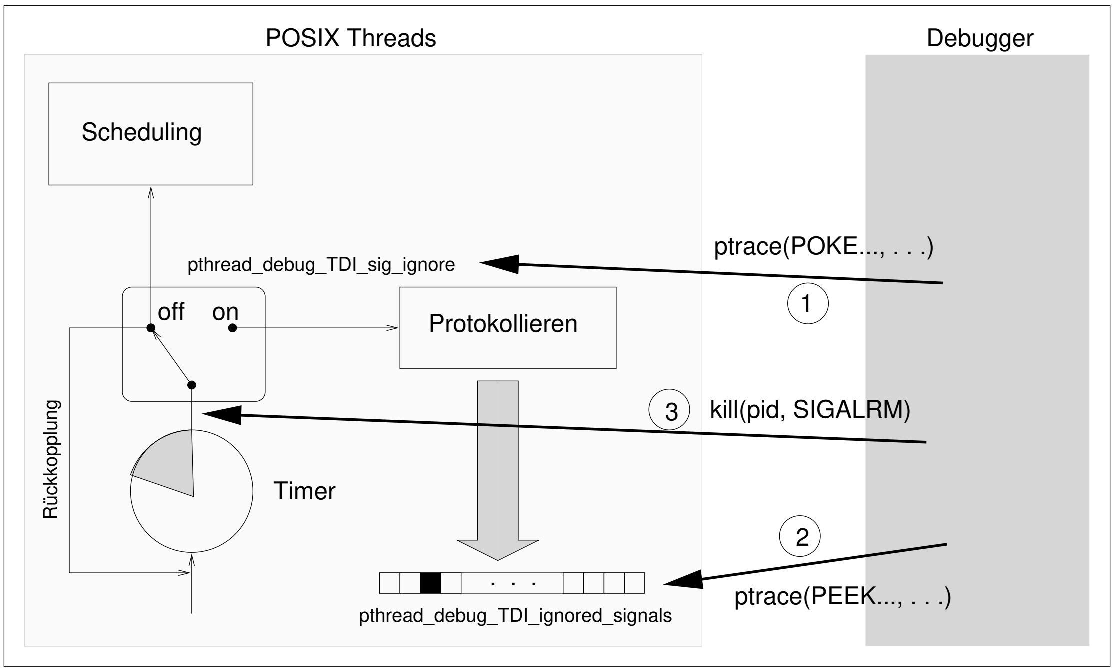

Abbildung 7.1.: Einflußnahme des Debuggers auf die Signalbehandlung

Fortführung des Schedulngs zu ermöglichen. Der Round-Robin-Schedulingalgorithmus ist meist so implementiert, das bei Ablauf eines Timers und damit bei Eintreffen des entsprechenden Signals (`SIGALRM`, `SIGVTALRM` oder `SIGPROF`) ein neuer Timer installiert wird, der das Ende des nächsten Zeitquantums bedeutet. Ein einmal verlorenes Signal hätte den Wechsel der Schedulingstrategie zur Folge, da der aktuelle Thread erst bei einem blockierenden Funktionsruf die CPU freigeben würde (FIFO). Um ein erneutes Senden der ignorierten Signale zu ermöglichen, müssen diese von der Pthreads-Implementierung protokolliert werden.

Der POSIX-Threads-Implementierung wurden die Symbole

- `pthread_debug_TDI_sig_ignore` und
- `pthread_debug_TDI_ignored_signals` hinzugefügt.

`pthread_debug_TDI_sig_ignore` wird für die Ausführung des TDI vom Debugger gesetzt. Nach der TDI-Ausführung enthält die Bitmaske `pthread_debug_TDI_ignored_signals` die Menge der eingetroffenen Signale. Ein mehrfaches Eintreffen eines Signales ist damit nicht mehr erkennbar. Dies bedeutet für das eigentliche Ziel der Vermeidung des signalbedingten Schedulungs keine Einschränkung, da bei Nichtbehandlung eines Timer-Signals keine weiteren Timer aktiviert werden und somit keine neuen Signale eintreffen können.

Die Bereitstellung der beiden Symbole und deren Benutzung in der POSIX-Threads-Implementierung ist optional. Es ist Aufgabe des Debuggers zu erkennen, ob diese globalen Symbole definiert sind.

7.1.1.2. Überladene Funktionen in MT-sicheren Bibliotheken

Die Programmierung mit dem POSIX-Threads-API unter Verwendung der Standard-C-Bibliotheken führt zu Problemen bei der Ausführung nicht reentranter Funktionen. Deshalb werden diese Funktionen häufig innerhalb der Pthreads-Implementierung durch reentrante Versionen überladen. Die dabei entstandene Implementierung nennt man multithread(MT)-sicher, da eine konsistente Multithread-Nutzung möglich ist. Weiterhin werden in User-Space-Implementierungen die Funktionen überladen, die ein Blockieren des Prozesses aufgrund von Ein-/Ausgabeoperationen hervorrufen können. Beispielsweise kann der Systemruf `read()` durch eine gleichnamige Version überladen werden, die prüft, ob Daten für den entsprechenden Dateideskriptor vorhanden sind. Sind keine Daten lesbar, so wird der Scheduler aktiviert und ein anderer Thread kann ausgeführt werden. Damit wird verhindert, daß der gesamte Prozeß aufgrund eines blockierenden Thread-Systemrufs gescheduled wird.

Während der Ausführung des TDI sind alle Threads gestoppt. Die TDI-Funktionen können ohne die Intervention anderer Threads ausgeführt werden. Die Benutzung nicht reentranter Funktionen ist möglich, sofern diese nicht auf aktuell von der Pthreads-Laufzeitumgebung genutzte Ressourcen zugreifen. Die Ausführung prozeßblockierender Ein-/Ausgaberoutinen ist im TDI-Kern ebenfalls unproblematisch, da während der TDI-Ausführung keinem anderen Thread CPU-Zeit zur Verfügung gestellt werden muß.

Benutzung alternativer Funktionen

Da weder reentrante noch spezielle Funktionen zur Ein-/Ausgabe benutzt werden müssen, sind für die TDI-Implementierung folgende Lösungen denkbar. Zum einen ist es möglich, alle überladenen Funktionen neu zu implementieren und im TDI-Kern unter anderem Namen zu verwenden. Die Daten, auf denen diese Funktionen arbeiten, dürfen nicht innerhalb der Pthreads-Implementierung sichtbar sein. Damit wird verhindert, daß sich TDI und Pthreads-Laufzeitumgebung hinsichtlich der verwendeten Daten beeinflussen. Statt der Neuimplementierung ist es auch denkbar, eine nicht reentrante Standard-C-Bibliothek zu modifizieren, indem alle Funktionsnamen geändert werden. Sie kann dann mit der Applikation gebunden und vom TDI benutzt werden. Diese Möglichkeit setzt jedoch voraus, daß die C-Bibliothek im Quellcode vorliegt.

Eine elegantere Lösung des Problems der überladenen Funktionen stellt die Möglichkeit dar, Funktionen erst zur Laufzeit zu binden. Viele UNIX-Systeme, die das ELF[1]-Binärformat unterstützen, bieten die von Solaris bekannte Funktion `dlopen` [Ope97] zum Laden gemeinsam benutzter Bibliotheken („shared libraries“) an. Mit der Funktion `dlsym` können zu bestimmten Symbolnamen die zugehörigen Adressen ermittelt und lokalen Zeigerobjekten zugewiesen werden. Es ist somit nicht mehr notwendig, die Funktionsnamen im Quellcode der überladenen Bibliothek zu ändern. Diese Lösung besticht durch ihren geringen Aufwand, hat jedoch auch Schwächen. Zum einen ist die Existenz der `dl`-Bibliothek, die die Funktionen für das dynamische Laden bereitstellt, nicht für jedes UNIX-System garantiert. Andererseits gibt es Probleme, wenn die `dl`-Funktionen von der Pthreads-Implementierung überladen werden. Hier müßte jedoch nur diese vergleichbar kleine Menge an Funktionen reimplementiert oder umbenannt werden.

7.1.1.3. Dynamische Speicherverwaltung

Das Laden von Funktionen nichtreentranter C-Bibliotheken zur Verhinderung des Scheduling ist nur erlaubt, wenn bei deren Ausführung keine Wechselwirkungen zwischen der Pthreads-Laufzeitumgebung und dem TDI-Kern entstehen. Hinsichtlich der Daten, die von den überladenen Funktionen der Pthreads und den dynamisch geladenen Funktionen des TDI benutzt werden, bestehen keine Abhängigkeiten, da sie unabhängig voneinander verwaltet werden. Abhängigkeiten entstehen jedoch dann, wenn prozeßspezifische Ressourcen gemeinsam benutzt werden.

Kernproblem sind die reentranten Funktionen zur Speicherverwaltung, wie sie beispielsweise in den FSU- und MIT-Pthreads-Implementierungen definiert werden. Die Semantik ist undefiniert, wenn der Speicher des Prozesses von beiden Seiten, der Pthreads-Implementierung und dem TDI, verwaltet wird. Speicherbereiche, die von einer Speicherverwaltung bereits vergeben wurden, werden in der anderen als frei deklariert. Für den TDI-Kern muß deshalb eine unabhängige Speicherverwaltung gewährleistet sein. In diesem Fall dürfen keine Funktionen der Standard-C-Bibliothek geladen werden.

Die hier erarbeitete Lösung basiert auf der Verwaltung eines statischen Speicherbereiches. Es wurden die Funktionen `tdi_malloc`, `tdi_realloc`, `tdi_calloc`, `tdi_free` und `tdi_strdup` zur Verfügung gestellt, die die Semantik der bekannten Standardroutinen haben.

7.1.2. Anfragebearbeitung

Für die Bearbeitung der TDI-QL-Anfragen wurde ein zweistufiger Mechanismus entwickelt. Er besteht aus einem Präkompiler, der die Anfrage in eine

[1] Das unter System V (Release 4) und Solaris bekannte Executable and Linking Format (ELF) bietet im Gegensatz zum alten a.out-Format eine einfachere Möglichkeit für das dynamische Laden von gemeinsam genutzten Bibliotheken.

Form umwandelt, die von der zweiten Stufe, dem Tupelinterpreter benutzt wird, um den Selektionsausdruck für jedes der in der entsprechenden Relation vorhandenen Tupel auszuwerten. Bei der Realisierung dieser Komponenten wurden hauptsächlich die GNU-Werkzeuge *flex* und *bison* als freie Vertreter der bekannten Scanner- beziehungsweise Parsergeneratoren *Lex* und *Yacc* verwendet [Sta95, Vol89, DS90, LMB92].

Um die Verfahrensweise besser zu beschreiben, soll hier die Beispielanfrage

```
,,<TED>thread:id,prio=10,state: (prio+10<20) && cvwf!=0x10''
```

betrachtet werden. Sie entspricht der verbalen Anfrage: ,,Gebe die Thread-ID, die Priorität und den Zustand aller Threads aus, deren Priorität um 10 erhöht kleiner als 20 ist und die nicht auf die Signalisierung der Bedingungsvariable mit der ID 10(hexadezimal) warten, und setze deren Priorität auf 10".

Der Präkompoiler hat die Aufgabe, die syntaktische Korrektheit der Anfrage zu überprüfen. Zusätzlich muß die statische Semantik des Ausdrucks analysiert werden. Fragestellung ist hierbei, ob die angegebenen Attribute auch im Relationenschema der spezifizierten Relation enthalten sind. Weiterhin lassen sich in der Anfrage Optimierungsmöglichkeiten erkennen. Zusammenfassend muß der Präkompiler die folgenden Informationen bereitstellen:

- Interne Bezeichnung (`RelType`) der spezifizierten Relation
- Liste der auszugebenden Attribute `Proj` mit ihrer internen Bezeichnung
- Tupelliste der zu schreibenden Attribute `Write` und ihrer Werte
- Sollen Duplikate ausgegeben werden ? (`SetOutput`)
- Welche Optimierungen sind möglich ? (`OptiList`)
- leicht evaluierbarer Selektionsausdruck `PreCompResult`

Für die Beispielanfrage erhielte man das Ergebnis:

- `RelType = ET_THREAD`,
- `Proj = <TA_ID, TA_PRIO, TA_STATE >`,
- `Write = <[TA_PRIO,10] >`,
- `PreCompResult = ''($TA_PRIO+10 <20) && $TA_CVWF != 0x10''` .

`ET_THREAD, TA_ID, TA_PRIO, TA_STATE, TA_CVWF` sind dabei Ganzzahlwerte des im TDI definierten Attribut-Enumerationstypen. Die Zeichenkette `PreCompResult` hat somit die folgende Gestalt `,,($2+10<20)&&$10!=0x10''`. Es soll die Menge der Werte ausgegeben werden, das heißt `SetOutput` hat den Wert `TRUE`. Optimierungen sind in der Anfrage nicht möglich, womit `OptiList` leer ist.

Mit den gewonnenen Informationen kann nun der Tupelinterpreter gestartet werden, dessen Kernalgorithmus in Abbildung 7.1.2 vereinfacht dargestellt ist. Die

```
Tupel := getFirst(RelType)
while (Tupel)
  do
    /* Selektionsprädikat auswerten */
    if (EvalSelect(RelType,Tupel,PreCompResult))
      then
        /* Ausgabe */
        if (SetOutput) /* Duplikate ? */
          then
            PrintOutProjAsSet (Tupel,Proj)
          else
            PrintOutProjAsIs  (Tupel,Proj)
          fi
        /* Zuweisung */
        WriteAttr    (Tupel,Write)
      fi
    Tupel = getNext(RelType,Tupel)
  done
```

Abbildung 7.2.: Algorithmus für die Anfrageauswertung

Kernfunktion des Algorithmus ist `EvalSelect`. Ihr werden Typ der Relation, derzeitiges Tupel der Relation sowie der vereinfachte Selektionsausdruck übergeben. Die Einträge `$<n>` im Selektionsausdruck `PreCompResult` werden durch die Auswertung der Attributlesefunktion für das `n`-te Attribut der spezifizierten Relation ersetzt. Die Ersetzung wird während des Parsens des vereinfachten Selektionsausdruckes durchgeführt. Ist für das entsprechende Attribut keine Lesefunktion vom TED registriert worden, so wird das Zeichen `'?'` für den *Null*-Wert in das Ergebnis eingefügt.

Abbildung 7.3 gibt einen Überblick der Anfragebearbeitung und skizziert die diesbezüglichen Beziehungen der Schnittstelle mit ihren Anrainern.

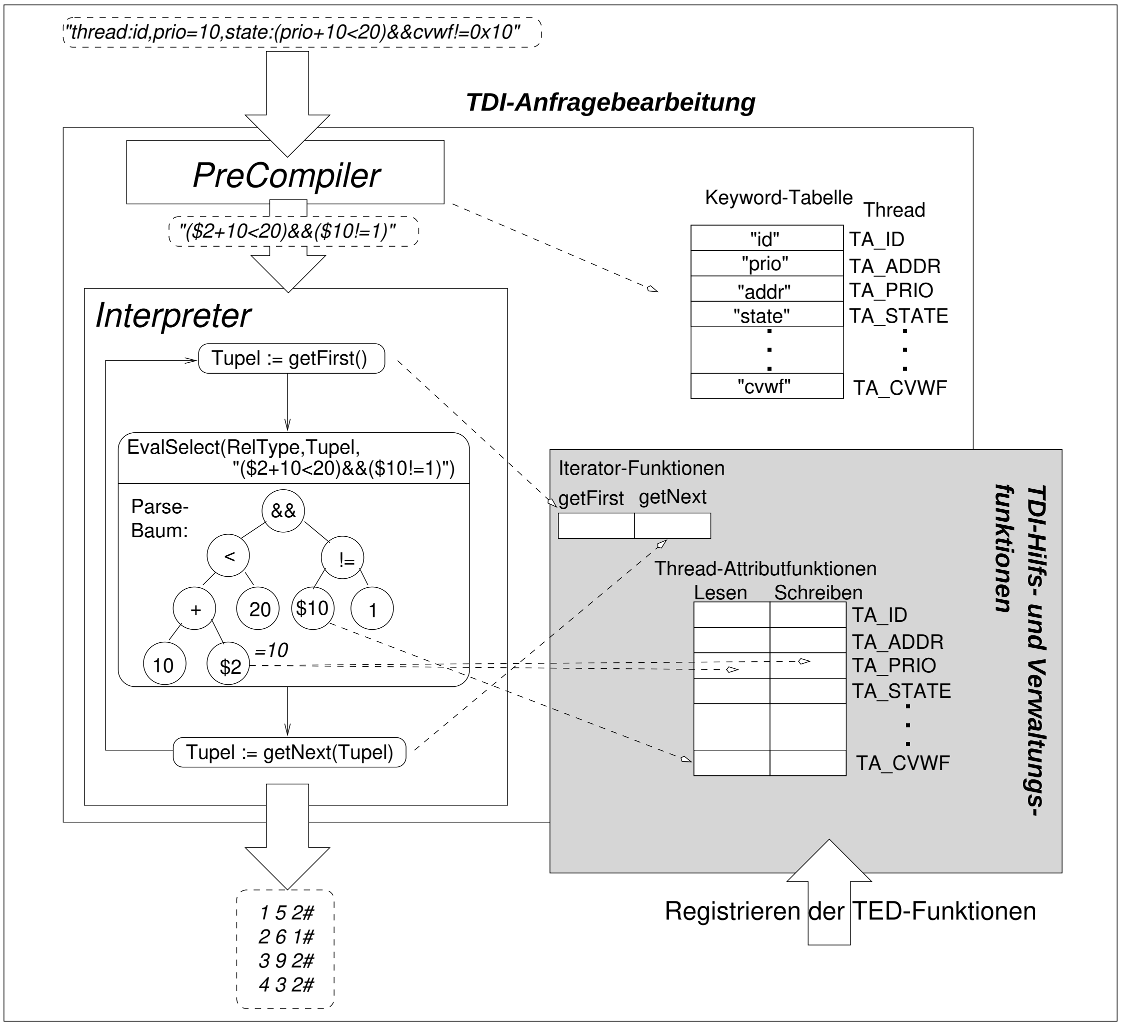

Abbildung 7.3.: TDI-QL-Anfragebearbeitung im Überblick

7.2. Anpassung der POSIX-Threads-Implementierungen

Bei der Anpassung der POSIX-Threads-Implementierungen sind folgende Aufgaben zu lösen. Eine Aufgabe besteht in der Implementierung der TED-Funktionen und deren Integration in die Pthreads-Implementierung. Ein weiterer großer Schwerpunkt liegt in der Verknüpfung von TDI und Pthreads-Implementierung. Hierzu gehören Funktionen für das dynamische Binden der TDI-Bibliothek sowie die Registrierung der TED-Funktionen beim TDI.

7.2.1. Implementierung der TED-Funktionen

Die Implementierung des TED stellt die Basis für die Bereitstellung des TDI dar. Es gibt zahlreiche POSIX-Threads-Implementierungen, sodaß eine Implementierung des TED exemplarisch nur für wenige Pthreads-Pakete vorgenommen wurde. User-Space-Implementierungen und Kernel-Thread-Lösungen wurden betrachtet. Bei der Realisierung der TED-Funktionen gibt es Pthreads-spezifische Unterschiede, die jedoch nicht auf die Einordnung der jeweiligen Implementierung in eine der zwei Gruppen zurückgeführt werden können. Deshalb wird hier keine Differenzierung hinsichtlich Kernel- oder User-Space-Threads vorgenommen.

Im Rahmen dieser Arbeit wurden TED-Implementierungen für die FSU-Pthreads, die MIT-Pthreads und die Linuxthreads geschaffen. Diese sind zum Teil unvollständig, demonstrieren jedoch die Durchführbarkeit der Anpassung. Bei der Implementierung gab es Besonderheiten, die hier erwähnt werden sollen.

7.2.1.1. Realisierung von Iteratoren

In der Thread-Erweiterung für das Debugging (TED) müssen Funktionen zur Mengeniteration bereitgestellt werden. Iteratoren für die Menge der Threads lassen sich leicht implementieren, da die Threadmenge von den Pthreads als Liste gespeichert wird, um das Scheduling zu realisieren oder um die Verwaltung des Stacks zu gewährleisten. TED-Module müssen Zugriff auf diese internen Datenstrukturen haben. Bei den drei erwähnten Pthread-Bibliotheken konnten die Iteratoren für die Menge der Threads sehr leicht realisiert werden, da alle Thread-Strukturen in verketteten Listen zugreifbar sind.

Sollten keine derartigen Listen zur Verfügung stehen oder ist die Einarbeitung in den Quelltext der Implementierung zu aufwendig, so können auch die vom TDI angebotenen Objektcontainer benutzt werden. Stellt die Pthreads-Implementierung keine persistenten Thread-ID's bereit, so ist die Benutzung der TDI-Objektcontainer ohnehin empfehlenswert.

Die Bereitstellung von Mengeniteratoren für die Objektklassen Mutex und Bedingungsvariable ist kaum über das Auswerten interner Pthreads-Daten möglich, da diese Objekte häufig nur dem Pthreads-Nutzer bekannt sind. Die Benutzung der TDI-Objektcontainer ist hier ebenfalls erforderlich.

Für die Registrierung der TED-Objekte durch das TDI sind Modifikationen der jeweiligen Pthreads-Implementierung notwendig. In den Routinen zur Erzeugung beziehungsweise zum Entfernen von POSIX-Threads-Objekten sollten Rufe der Registrierfunktionen integriert werden. Die Funktionen, in denen die Rufe plaziert werden müssen, sind `pthread_create` und `pthread_exit` für Threads sowie die `pthread_..._init`- und `pthread_..._destroy`-Routinen für Mutexe und Bedingungsvariablen. Selbstverständlich muß an allen Stellen der Benutzung der

Objektcontainerfunktionen getestet werden, ob das TDI gebunden und die Funktionszeiger auf die TDI-Funktionen für die Objektverwaltung gesetzt sind.

7.2.1.2. Attributlesefunktionen

Alle zu implementierenden Lesefunktionen haben die Gestalt

```
AttrDomainType get<RelName><AttrName>(ObjRefT Obj)
```

und erhalten im Argument einen Zeiger auf ein Objekt der entsprechenden Relation. Für Attribute, die in der von der Pthreads-Implementierung definierten Objekt-Struktur enthalten sind, wird die Lesefunktion einfach durch Selektion des entsprechenden Strukturelements mit `Obj->Attr` realisiert.

Bei synthetischen Attributen ist die Realisierung der Lesefunktion aufwendiger. Dies soll am Beispiel der Relation `Thread` und dem Attribut `cvwf` erläutert werden. Um die Bedingungsvariable `CV` anzugeben, auf deren Signalisierung der Thread `T` wartet, muß die Menge der Bedingungsvariablen durchsucht werden. Meist sind Zeiger auf die Warteschlangen der Threads, die auf eine Bedingungsvariable blockieren, in der Struktur der Bedingungsvariable selbst enthalten. Wird der Thread `T` in der Warteschlange von `CV` gefunden, so kann die Suche abgebrochen werden. Die ID der Bedingungsvariable `CV` kann als Ergebnis der Funktion `getThreadCVWF` zurückgegeben werden. Die Suche wird beim ersten Auftreten des Threads `T` abgebrochen, weil ein Thread zu einem bestimmten Zeitpunkt nur auf eine einzige Bedingungsvariable warten kann.

Ein Attribut, das für das Setzen von Breakpoints benötigt wird, ist das Thread-Attribut `newpc`. Der Wert von `newpc` wird vom Thread-Debugger benötigt, um Kontextwechsel zu erkennen oder einen beliebigen Thread an der nächstmögichen Stelle zu stoppen (siehe Abschnitt 8.6.1). Der Wert von `newpc` gibt die Code-Adresse an, bei der der Thread seine Ausführung fortsetzt. Bei User-Space-Implementierungen, deren Scheduling auf der Benutzung von `longjmp` und `setjmp` beruht, ist der Wert von `newpc` aus dem mit `setjmp` erzeugten Kontextblock lesbar [POSb, POSa]. Der Wert von `newpc` ist für den aktuellen Thread nicht ermittelbar, da er erst beim nächsten Kontextwechsel geschrieben wird. Bei Kernel-Threads-Implementierungen können die Registerwerte der einzelnen Threads über Systemschnittstellen wie das `/proc`-Dateisystem gelesen werden.

7.2.1.3. Attributschreibfunktionen

Das Schreiben von Attributwerten kann im Vergleich zu den Lesefunktionen mit einem Mehraufwand verbunden sein. Da die Attributwerte nur den Zustand eines POSIX-Thread-Objektes anzeigen, führt ein Schreiben des Wertes nicht unbedingt zu einer Zustandsänderung. Im allgemeinen sind komplexe Operationen nötig, um den Objektzustand zu ändern.

Am Beispiel des Attributes `state` wird die Komplexität deutlich. Das Schreiben des `state`-Attributes entspricht der Kontrolle des Schedulings, wie in Abschnitt 6.2.1 (Seite 27) bereits beschrieben wurde. Um einen Thread zu aktivieren oder zu stoppen, sind Umordnungen in den Thread-Warteschlangen erforderlich.

Für einige Attribute der TED-Objekte ist es nicht sinnvoll, diese zu schreiben. Ein Schreibschutz für Attribute läßt sich einfach dadurch erreichen, daß für dieses Attribut keine Schreibfunktion registriert wird. Durch die getrennte Registrierung von Schreib- und Lesefunktionen läßt sich natürlich auch ein Leseschutz erreichen. Andere Attribute können zwar geschrieben werden, jedoch nur mit bestimmten Werten. Beispielsweise ist das Schreiben des Thread-Attributes `state` mit dem Wert $\texttt{BLOCKED}_{\texttt{MUTEX}}$ wenig sinnvoll, da eine derartige Zustandsüberführung nicht realisiert werden kann.

7.2.2. Verknüpung von Pthreads und TDI

Neben der Bereitstellung der TED-Funktionen müssen in der POSIX-Threads-Implementierung Maßnahmen zur Einbindung des TDI getroffen werden. Wesentliche Aufgaben sind das Integrieren des TDI-Moduls bei gewünschter TDI-Unterstützung sowie die Registrierung der TED-Funktionen.

7.2.2.1. Einbinden des TDI-Moduls

Das Einbinden des TDI-Moduls kann zur Laufzeit oder in der Linker-Phase der Programmerzeugung erfolgen. Der Entwickler der POSIX-Threads-Implementierung kann entscheiden, welche Methode benutzt wird. Für die Implementierung des Debuggers ist es unerheblich, auf welche Weise die TDI-Bibliothek mit der Applikation gebunden wird. Seine in Abschnitt 6.6.1 beschriebene TDI-Perspektive ändert sich nicht.

Wird im folgenden von Compiler oder Linker gesprochen, so sind damit die bekannten GNU-Werkzeuge **gcc** und **ld** gemeint. Diese Werkzeuge sind für alle in dieser Arbeit betrachteten Plattformen verfügbar [Cha94, Sta91].

Statisches Binden

Die Einbindung durch das Linken der TDI-Bibliothek zur Compile-Zeit erfordert für den Nutzer die Angabe eines weiteren Argumentes für den Linker, hier die Angabe der Bibliothek `libTDISrv.a` . Für das Linken der Bibliothek ist es weiterhin erforderlich, der Applikation eine Benutzung der `TDI_Server`-Funktion hinzuzufügen. Dazu muß im Quelltext der Applikation ein Ruf der Funktion enthalten sein, der jedoch in keinem Fall ausgeführt wird, beispielsweise:

```
if(0) TDI_Server(NULL);
```

Dies ist notwendig, da sonst weder in der Applikation noch in der Pthreads-Implementierung ein Ruf dieser Funktion stattfindet. Wird der Ruf der Funktion `TDI_Server` nicht eingefügt, stuft der Linker die `libTDISrv.a`-Bibliothek als ungenutzt ein und bindet sie nicht mit der Applikation.

Nachteilig am statischen Binden der TDI-Bibliothek sind die damit verbundenen umfangreichen Vorbereitungen für die Nutzung des TDI. Für das Debugging einer gewöhnlichen C-Applikation muß der Nutzer lediglich die Compileroption „-g“ angeben, um die für das Debugging wichtigen Symbolinformationen vom C-Compiler generieren zu lassen. Das Angeben der TDI-Bibliothek in einem weiteren Compiler-Argument ist daher nur ein geringfügiger Mehraufwand für den Nutzer. Anders verhält es sich mit dem Einfügen des Rufes der Funktion `TDI_Server`. Der Nutzer muß diesen Ruf bei gewünschter TDI-Nutzung einfügen, ihn jedoch wieder entfernen, wenn die Applikation nicht debugged wird. Das Entfernen des Rufes ist notwendig, da bei nichtgebundener Bibliothek `libTDISrv.a` ein Linker-Fehler auftritt.

Dynamisches Binden

Eine Alternative zum statischen Einbinden des TDI ist das dynamische Binden zur Laufzeit. Hierzu wird das TDI-Modul in einer Form bereitgestellt, die das dynamische Binden erlaubt (`libTDISrv.so`).

Mit Hilfe der erwähnten Funktion `dlopen` (siehe Abschnitt 7.1.1.2, S.59) ist es möglich, die Bibliothek `libTDISrv.so` während der Ausführung der Applikation zu laden. Die Adresse der TDI-Funktion `TDI_Server` wird mit der Funktion `dlsym` ermittelt und einem in der Pthreads-Implementierung definierten Funktionszeiger gleichen Namens zugewiesen.

Das Laden der Bibliothek wird abhängig vom Wert des in Abschnitt 6.6.1 erwähnten Symbols `pthread_debug_with_TDI` durchgeführt, das in der Pthreads-Implementierung definiert sein muß. Der Debugger kann diese globale Variable lesen und zu Beginn der Ausführung der Applikation auf einen definierten Wert setzen, der ein Debugging signalisiert.

Das dynamische Laden der TDI-Bibliothek ist mit den bereits in Abschnitt 7.1.1.2 erwähnten Nachteilen verbunden. Diese Lösung hat jedoch eine Reihe von Vorteilen. Zum einen ist es nicht mehr notwendig, die Quellen der Pthreads-Applikation im Hinblick auf das angesprochene Linker-Problem um einen Funktionsruf zu ergänzen. Es muß lediglich die Compiler-Option „`-ldl`“ angegeben werden, die später dem Linker übergeben wird. Diese Option veranlaßt das Laden der Bibliothek `libdl.so`, die die `dl...`-Funktionen enthält. Bei einer Pthreads-Bibliothek, die dynamisches Binden verwendet, muß diese Option auch angegeben werden, wenn keine Debug-Unterstützung gefordert wird. In diesem Sinne stellt die Angabe der Compileroption keine spezielle Vorbereitung für das Debugging mit dem

TDI dar, sondern ist ein notwendiger Schritt bei der Verwendung der POSIX--Threads-Bibliothek. Ein weiterer Vorteil des dynamischen Bindens ist, daß die erzeugte Applikation auch nach der Entwicklungsphase benutzt werden kann, ohne neu erzeugt zu werden. Die TDI-Funktionalität wird nur gebunden, wenn dies durch den Debugger signalisiert wurde.

7.2.2.2. Registrieren der TED-Funktionen

Das Registrieren der TED-Funktionen (siehe Abschnitt 6.5.1) muß vor dem Erzeugen der Threads und dem Initialisieren der Mutexe und Bedingungsvariablen geschehen. Für die Plazierung der diesbezüglichen Operationen bietet sich deshalb eine initial ausgeführte Funktion (in einigen Implementierungen `pthread_init`) an. Diese Funktion sollte vor der Ausführung der für die Erzeugung und Initialisierung von Objekten benutzten Funktionen gerufen werden, da nur so eine Registrierung aller jemals instantiierten Objekte möglich ist.

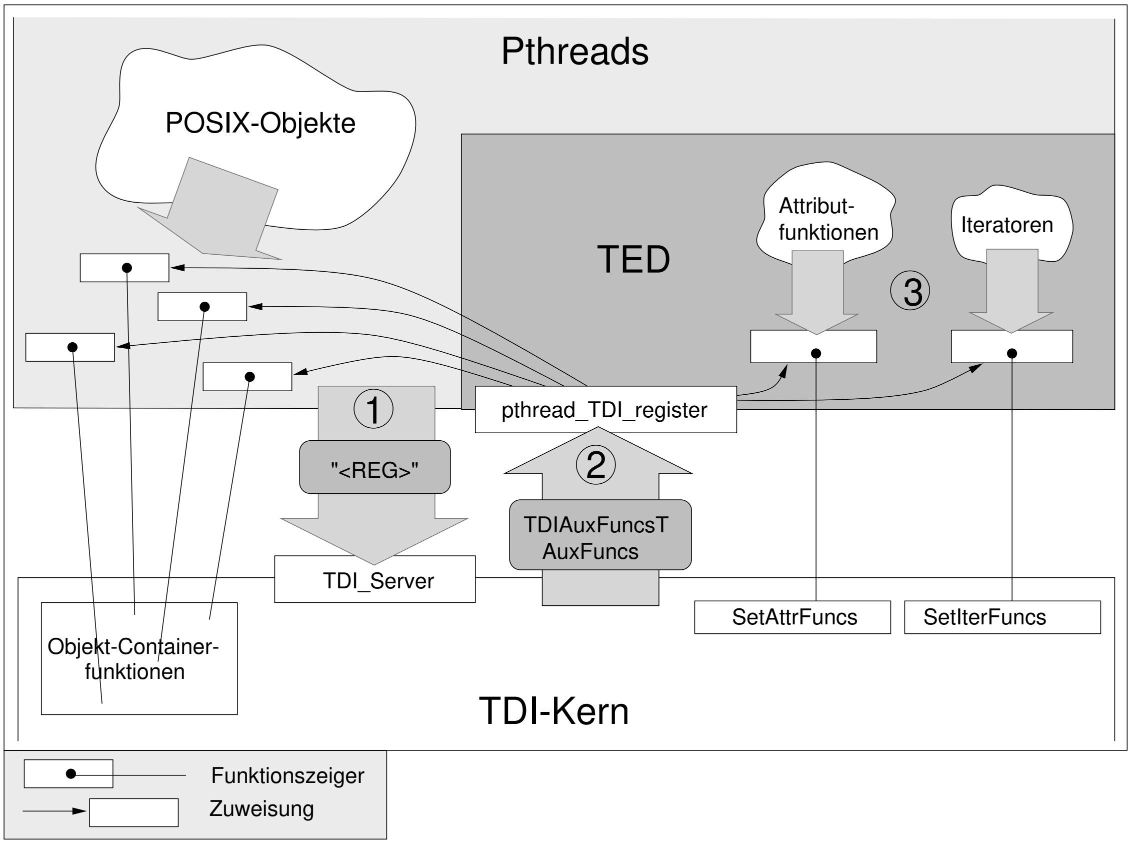

Abbildung 7.4.: Verknüpfen des TED mit dem TDI-Kern

Das Registrieren der TED-Funktionen kann nur über die vom TDI zur Verfügung gestellten Funktionen geschehen. Diese TDI-Funktionen müssen dem TED-Modul durch Aufruf der Funktion `pthread_TDI_register` bekannt gemacht werden. In der Abbildung 7.4 sind die einzelnen Schritte der TDI-Einbindung beschrieben.

Um den TDI-Kern zu veranlassen, seine Registrierfunktionen bekanntzugeben, wird die Funktion `TDI_Server` mit dem Argument `,,<REG>''` aufgerufen. Daraufhin werden die für das TED benötigten Funktionen in einer Struktur des Typs `TDIAuxFuncsT` mit dem Ruf von `pthread_TDI_register` an das TED-Modul übergeben. Im TED müssen Funktionszeiger definiert sein, denen die in der Struktur `TDIAuxFuncs` enthaltenen Funktionsadressen zugewiesen werden können. Der in der Abbildung beschriebene dritte Schritt bezeichnet die eigentliche Registrierung der TED-Funktionen. Hierfür werden die Funktionszeiger benutzt, die die Adressen der TDI-Funktionen `SetAttrFuncs` und `SetIterFuncs` enthalten.

8. Thread-Debug-Erweiterung des Gdb

Die Thread-Debug-Schnittstelle bietet eine komfortable Möglichkeit des Zugriffs auf die Daten der POSIX-Threads-Implementierung. Befehle für das Thread-Debugging können unter Verwendung des TDI auf einfache Weise implementiert werden.

Um die entstehenden Möglichkeiten der TDI-Nutzung aufzuzeigen, wurde exemplarisch ein existierender Debugger im Hinblick auf das TDI angepaßt. Die „Wahl" fiel dabei auf den GNU Debugger GDB [Sta89], der zum Zeitpunkt des Beginns dieser Arbeit in der Version 4.17 vorlag. Der GDB ist wohl einer der meistverwendeten Debugger in der UNIX-Welt, sodaß die Änderungen in Bezug auf das Thread-Debugging einer großen Nutzerschaft dienen könnten. Die Vorteile des GDB liegen unter anderem in der freien Zugänglichkeit des Quelltextes[1], der weiten Verbreitung, der Lauffähigkeit auf den verschiedensten UNIX-Plattformen und in seiner überragenden Leistungsfähigkeit im Bereich des Low-Level-Debugging. Bedingt durch die frei verfügbaren Quellen gibt es eine große Entwicklergemeinschaft, die eine zügige und vor allem nutzerorientierte Entwicklung des GDB ermöglicht.

Dieses Kapitel soll veranschaulichen, wie die Funktionen eines herkömmlichen Breakpoint-Debuggers und die des TDI kombiniert werden können, um einen Thread-Debugger zu schaffen. Obwohl am Beispiel des GDB erläutert, sind viele Änderungen auch auf andere Breakpoint-Debugger übertragbar.

8.1. Funktionalität des Thread-Debuggers

Im Abschnitt 4.1.3 wurden bereits die Schwächen eines herkömmlichen Debuggers in Bezug auf die Anwendbarkeit für Multi-Thread-Anwendungen beschrieben. Folgende Möglichkeiten sollte ein Thread-Debugger bieten:

i) Auflistung aller vorhandenen Threads, Mutexe und Bedingungsvariablen,

[1] Wie alle GNU-Werkzeuge unterliegt auch der GDB den Bestimmungen der GNU General Public License (GPL).

ii) Erkennung, Selektion sowie Bindung eines Thread-Kontrollflusses,

iii) Anzeige von Kontextwechseln und Eingriff in das Scheduling.

Einige der in Kapitel 4.2 vorgestellten verwandten Arbeiten beschäftigen sich mit der Erkennung fehlerhafter Programmzustände in Thread-Anwendungen (Smart-GDB [Hal96], MpD [PHK91], KDB [BKS96]). Unter anderem können Deadlocks oder die Verletzung des gegenseitigen Ausschlusses beim Datenzugriff (Data Race) erkannt werden. Betrachtet man die Lösungen, so fällt die Möglichkeit auf, diese Funktionalität auf der Grundlage des TDI zu implementieren. Der hier vorgestellte Debugger auf der Grundlage des GDB soll ausdrücklich die Basisfunktionalität des Thread-Debuggings bezüglich der Pthreads-Anwendungen bieten, die zur Umsetzung von High-Level-Debuggern nötig ist. Somit können bestehende High-Level-Debugger, die an bestimmte Debugger- sowie Pthreads-Implementierungen gebunden sind, durch die Portierung auf das TDI für alle vom TDI unterstützten Pthreads-Implementierungen eingesetzt werden.

8.2. Informationen zu Pthreads-Objekten

Die Bereitstellung von Informationen über die in der POSIX-Threads-Laufzeitumgebung instantiierten Objekte ist eine der Hauptaufgaben des TDI, sodaß die entsprechenden Befehle im wesentlichen durch Benutzung des TDI realisiert werden können. Im Debugger müssen dazu lediglich die mit dem TDI gewonnenen Ergebnisdaten in einer für den Nutzer ansprechenden und nützlichen Weise angezeigt werden.

8.2.1. Allgemeine Informationen

Um das Anzeigen der Daten für Threads, Mutexe und Bedingungsvariablen zu ermöglichen, wurde der **info**-Befehl des GDB erweitert. Folgende Varianten wurden hinzugefügt:

info pthreads Anzeigen aller vorhandenen (POSIX-)Threads mit ihren Status-Informationen

info pmutex Anzeigen aller instantiierten Mutexe

info pcond Anzeige der vorhandenen Bedingungsvariablen

Der folgende vereinfachte Programmausschnitt beschreibt die Funktionsweise des **info pthreads** -Befehls:

```
...

/* Senden der Anfrage */
TDI.send(&TDI,"<TED>thread: id,addr,prio,state,"
              "rstate,entry,earg,mbo,cvwf");

/* Signalisierung des TDI-Kerns */
Target.signal();

/* Empfangen des Ergebnisses */
TDI.receive(&TDI,Ergebnis);

/* Aufteilung in Tupel */
while (tupel=getTupel(Ergebnis)) {

  /* Einlesen der Atribute des Tupels */
  sscanf(tupel,"%x%x ... %x",idVal, ..., cvwfVal);

  /* Ausgabe */
  GDB.out("%d ... %d",idVal, ... , cvwfVal);

}
```

In der ersten Phase wird die Anfrage an das TDI gesendet. Da möglichst alle Thread-Objekte und deren Werte ermittelt werden sollen, wird kein Selektionsprädikat angegeben. Weiterhin beinhaltet die Menge *Proj* alle Attribute der Thread-Relation. `TDI` bezeichnet eine instantiierten TDI-Klienten.

Um die Anfragebearbeitung anzustoßen, muß der TDI-Kern signalisiert werden. Wie bereits erwähnt, kann dies mit Hilfe eines UNIX-Signals oder durch den Aufruf der `TDI_Server` Funktion im Zielprozeß erfolgen. Der erste Fall kann mit einem `kill()`-Aufruf abgedeckt werden. Für das Rufen einer Funktion im Ziel-Prozeß steht im GDB die Funktion `parse_and_eval()` bereit, der als Argument ein C-Ausdruck übergeben wird, dessen Auswertung auf Grundlage von Zielprozeßdaten erfolgt. Ist ein Funktionsaufruf Bestandteil des C-Ausdruckes, so wird die entsprechende Funktion im Zielprozeß ausgeführt. Der Ruf `parse_and_eval("TDI_Server(\"SV_IPC_SM\")")` bewirkt das Kopieren des Argumentes `"SV_IPC_SM"` auf den Stack des derzeit aktiven Threads und den anschließenden Aufruf der Funktion `TDI_Server` im Zielprozeß.

Das Anfrage-Ergebnis wird mit der `receive`-Methode des TDI-Klienten gelesen. `Ergebnis` ist eine durch '`#`' getrennte Tupelliste. Jede Tupel-Zeichenkette muß wiederum in die einzelnen Bestandteile zerlegt werden. Diese müssen dann formatiert ausgegeben werden.

Bei der Erklärung der weiteren GDB-Befehle wird eine verkürzte Schreibweise benutzt; Senden, Signalisieren und Empfangen werden wie folgt zusammengefaßt: `<Ergebnis>=TDI.request(<Anfrage-String>)`.

8.2.2. Nutzerdefinierte Anfragen mit TDI-QL

Die neuen **info**-Befehle des GDB sollen den Standardzugang zu den POSIX-Threads-Informationen darstellen. Die Möglichkeiten des TDI werden dabei jedoch nicht in vollem Umfang genutzt. Dem Nutzer soll deshalb zusätzlich die Möglichkeit eingeräumt werden, die Anfragesprache des TDI direkt zu verwenden. Der entsprechende GDB-Befehl hat die folgende Syntax:

tdi <*Anfrage-String*> - Senden der Anfrage an das TDI und Ausgabe des Ergebnisses.

Die Implementierung ist ähnlich der der **info**-Befehle, der <*Anfrage-String*>-Parameter wird jedoch direkt an das TDI übergeben. Die Ausgabe des Ergebnisses erfolgt derzeit noch rudimentär, indem einfach die Ergebnis-Tupel so ausgegeben werden, wie sie vom TDI-Kern empfangen wurden. Hier liegen noch Möglichkeiten der Verbesserung, die im folgenden anhand eines Beispiels skizziert werden sollen.

Gegeben sei die Anfrage `"thread:id,entry,state:state==1 || mbo==0"`, die im Ergebnis alle Threads mit ihrer ID, der Startfunktionsadresse und dem Zustandswert enthält, die nicht auf einem Mutex blockiert sind oder deren Zustand `RUNNING` ist. Bereits die Anfragezeichenkette offenbart einige Schwächen. Der Nutzer muß Kenntnis darüber haben, daß der Zustand `RUNNING` mit `1` und die Tatsache, daß ein Thread nicht auf einem Mutex blockiert ist, mit `0` kodiert wird. Hier wäre es sinnvoll, einen Präprozessor einzusetzen, der Nutzereingaben wie `RUNNING` oder `NO_MUTEX` in die entsprechenden Zahlenwerte übersetzt. Weiterhin wäre es vorstellbar, daß Symbolnamen von Variablen und Funktionen stellvertretend für ihre Werte angegeben werden. Bei der Ausgabe des Ergebnisses sollten korrespondierend zu den projezierten Attributen deren Ausgabeformate eingehalten werden. Eine Ausgabe der Beispielanfrage könnte wie folgt aussehen:

```
8 804b238 2 #7 804b238 2 #10 804b238 1.
```

Unkomfortabel für den Nutzer ist, daß alle Zahlenwerte hexadezimal dargestellt werden, was bei der ID des Thread 16 (0x10) zu einer Fehlinterpretation führen könnte. Die angesprochene Umwandlung von Symbolnamen in Symbolwerte bei der Anfrage sollte bei der Ausgabe des Ergebnisses invers ausgeführt werden. So könnten die Funktionsadressen in Funktionsnamen und die Zustandswerte in die entsprechend lesbareren Zeichenketten `RUNNING`, `READY` und so fort transformiert werden.

Anhand des Beispiels wird deutlich, daß der Debugger einen Prä- und Postprozessor zur Bearbeitung der Anfragen und Ergebnisse bereitstellen sollte, um den TDI-QL-Zugang komfortabler zu machen. Diese Funktionalität könnte zum Teil vom Debugger und vom TDI-Klienten bereitgestellt werden. Für konstante Werte, wie sie zur Zustandskodierung im TDI definiert sind, könnten auch im TDI die entsprechenden Zeichenketten für die Ausgabe zur Verfügung gestellt werden, um die Anfrageergebnisse vergleichbar zu machen. Die Umwandlung von Symbolnamen in Werte und umgekehrt kann nur im Debugger geschehen, da er die Funktionalität zum Lesen der Symbolinformationen aus den Objekt-Dateien besitzt.

8.2.3. Anzeige der Ausführungsstacks

Die Angabe des Ausführungsstacks eines Threads ist für den Nutzer eine der wichtigsten Debugging-Informationen. Oft ist die Kenntnis der Aufrufkette ausreichend, um ein fehlerhaftes Verhalten zu erkennen. Ist man in der Lage, die Ausführungsstacks aller Threads zu betrachten, so lassen sich beispielsweise Data-Races erkennen, bei denen Daten geschrieben werden, ohne daß die entsprechenden Mutexe gelockt wurden. Sehr hilfreich ist die Sichtbarkeit aller Thread-Stacks, um eine Verklemmungssituation (Deadlock) zu analysieren.

Die aktuell ausgeführte Funktion und der Ausführungsstack eines Threads sind mit den Werten für Programmzähler (`pc`) und Stackpointer (`sp`) berechenbar. Die aktuelle Funktion wird anhand des `pc` ermittelt. Der Ausführungsstack wird von der Adresse `sp` an traversiert. Im Stack sind die Rückkehr-Adressen der rufenden Funktionen, lokale Variablen und die Parameterwerte der Funktionen kodiert. Der GDB ist in der Lage, diese Information auf der Grundlage von `pc` und `sp` zu ermitteln und darzustellen. Hierzu wird das Kommando **backtrace** angeboten.

Ein Thread-Debugger muß in der Lage sein, die Ausführungsstacks aller instantiierten Threads anzuzeigen. Der Stack des aktuellen Threads ist einfach darzustellen, da er mit `pc` und `sp` des Prozesses ermittelt werden kann. Um die Stacks der anderen Threads zu ermitteln, ist es lediglich notwendig, deren `pc`- und `sp`-Werte zu ermitteln. Die Thread-Relation bietet dazu die Attribute `newpc` und `sp`.

Dem GDB wurde der Befehl **ptstack** hinzugefügt, der den Stack eines bestimmten POSIX-Threads anzeigt. Er wurde vereinfacht wie folgt realisiert:

```
ptstack_command(<id>) {
  ...
  /* TDI-Anfrage (vereinfacht) */
  sscanf(TDI.request("<TED>thread:newpc,sp:id==<id>"),
         "%d %d",&pcVal,&spVal));

  /* Selektieren des Stacks */
  set_current_frame(pcVal,spVal);
```

```
 /* Ausühren des backtrace-Kommandos */
 execute_command("backtrace");

 /* Selektion des derzeitigen Stacks */
 get_current_frame(readpc(),readsp());
}
```

In der ersten Phase der Befehlsausführung werden die Register `pc` und `sp` des mit <`id`> spezifizierten Threads unter Verwendung des TDI ermittelt. Das entsprechende Stack-Segment wird dann mit `set_current_frame` selektiert. Der GDB-Befehl **backtrace** liest das derzeit selektierte Stacksegment und traversiert den Aufrufstack abwärts. Dabei erfolgt die Ausgabe der Aufrufkette des Threads. Um die Ausführung weiterer **backtrace**-Kommandos zu ermöglichen, muß erneut das oberste Stacksegment des aktuellen Threads ausgewählt werden.

8.3. Bindungsmöglichkeiten für Thread-Befehle des GDB

Wesentliche Aufgabe eines Thread-Debuggers ist es, dem Nutzer die Möglichkeit zu geben, einzelne Threads respektive ihre Kontrollflüsse zu betrachten. In Abschnitt 4.1.3 wurden die Probleme beschrieben, die sich hier mit einem prozeßgebundenen Debugger ergeben.

Grundsätzlich sollten alle Befehle des GDB, die sich auf den Kontrollfluß eines Prozesses beziehen, auch für einen spezifischen Thread-Kontext, das heißt threadgebunden, anwendbar sein. In dieser Arbeit werden zwei Möglichkeiten der Threadbindung vorgeschlagen:

a) Die Threadbindung für die Ausführung eines Kommandos und

b) die globale Bindung der Kommandos an einen Kontrollfluß.

Die Threadbindung eines einzelnen Befehls (**a)**) ist dann vorteilhaft, wenn nur kurze Phasen der Ausführung eines bestimmten Threads betrachtet werden sollen. Ist beispielsweise ein Breakpoint von einem Thread T erreicht worden, so könnte der Nutzer daran interessiert sein, welche Zeile als nächstes durch T ausgeführt wird. Hier ist ein threadgebundenes **next** hilfreich, das erst dann terminiert, wenn die nächste Quellzeile durch den Thread T erreicht wurde. Ein ungebundenes **next** würde auch dann zurückkehren, wenn die nächste Quellzeile von einem anderen Thread erreicht ist. Nach der Ausführung eines threadgebundenen Einzelbefehls sind wieder alle Thread-Kontrollflüsse sichtbar.

Die in **b)** angesprochene globale Threadbindung der Debugging-Befehle ermöglicht die Verfolgung eines einzelnen Kontrollflusses. Ist die Funktion `f()` die Startfunktion mehrerer Threads, so kann es für den Nutzer von Interesse sein, die Ausführung von `f()` durch einen ausgezeichneten Thread T zu beobachten. Zu diesem Zweck können alle GDB-Befehle an den Kontrollfluß des Threads T gebunden werden.

Im folgenden wird die Realisierung der beiden Bindungsvarianten beschrieben.

8.4. Threadbindung von Einzelbefehlen

Befehle eines Breakpoint-Debuggers, die mit der Ausführung der Zielapplikation zusammenhängen, lassen sich wie folgt charakterisieren. Die Ausführung eines jeden Befehls bewirkt, daß die Zielapplikation bis zum Erreichen eines Stopp-Kriteriums ausgeführt wird. Dabei kann angegeben werden, ob eine Instruktion (**stepi, nexti**), eine Quellzeile (**step, next**) oder ob die Applikation bis zum Erreichen des nächsten nutzerdefinierten Breakpoints ausgeführt werden soll (**continue**). Erfüllt der Zielprozeß das angegebene Kriterium, so stoppt er und die Kontrolle geht an den Debugger über.

Beim Debugging einer Multithread-Applikation unter Verwendung eines prozeßgebundenen Debuggers kann es vorkommen, daß der Thread-Kontext, der Ausgangspunkt eines der oben genannten Befehle war, nicht mit dem Thread-Kontext übereinstimmt, in dem die Applikation nach der Ausführung stoppt. Eine Threadbindung der Befehle bedeutet, daß der Thread-Kontext zum Zeitpunkt des Befehlsstarts mit dem übereinstimmt, der beim Programmstopp erreicht wird.

Im folgenden soll beschrieben werden, wie für einzelne Befehle des GDB eine Threadbindug realisiert wurde.

8.4.1. Threadspezifische Breakpoints

Das Stoppen eines bestimmten Threads bei Erreichen eines Breakpoints kann nur realisiert werden, wenn es möglich ist, threadspezifische Breakpoints zu setzen. Deshalb soll zunächst die Realisierung eines threadspezifischen **break**-Kommandos beschrieben werden. Ein threadspezifisches **continue** für einen Thread T kann dann leicht realisiert werden, indem alle gesetzten Breakpoints an den Thread T gebunden werden.

Der GDB bietet bereits die Möglichkeit, threadspezifische Breakpoints zu definieren. Der Befehl **break** hat die folgende Syntax:

break <*Location*> [**thread** <*TID*>].

Die Semantik des Befehls ist, daß die Zielapplikation nur gestoppt wird, wenn der Thread mit der ID <*TID*> die mit <*Location*> adressierte Instruktion ausführt. Die Angabe von **thread** ist derzeit schon für einige Kernel-Thread-Varianten möglich, zu denen unter anderem die Solaris-Threads, die Kernel-Threads unter

Mach und die Linux-Kernel-Threads zählen. Es kann nicht ausgeschlossen werden, daß eine Pthreads-Applikation auf einem der genannten Systeme verwendet wird. Um Konflikte beim Debugging von Kernel-Threads und POSIX-Threads zu vermeiden, werden threadspezifische Breakpoints für POSIX-Threads mit dem Schlüsselwort **pthread** erzeugt. Daraus ergibt sich folgende geänderte Syntax des **break**-Befehls:

break <*Location*> [**thread** <*TID*>] [**pthread** <*POSIX-TID*>].

Die Realisierung des threadgebundenen **break**-Befehls erfordert die Kenntnis einiger GDB-Details. Die Funktionen `proceed`, `wait_for_inferior`, `resume` und `normal_stop` stehen dabei im Mittelpunkt. Die Abbildung 8.1 zeigt den Aufrufgraphen dieser GDB-Funktionen. Die Funktion `proceed` wird von allen Kontroll-

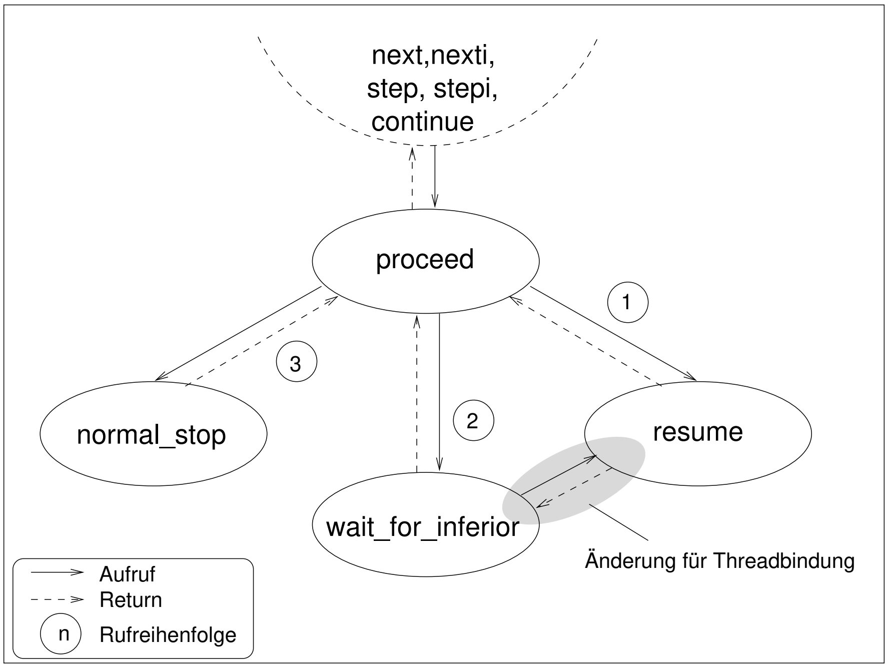

Abbildung 8.1.: Aufrufgraph der GDB-Funktionen zur Ausführung des Zielprozesses

flußbefehlen des GDB benutzt, um die Applikation auszuführen. `proceed` ruft zuerst die Funktion `resume` auf, um die Ausführung des Zielprozesses anzustoßen. Die Funktion `wait_for_inferior` führt den Systemruf `wait` aus, um auf ein Stoppen der Applikation zu warten. `normal_stop` bereitet die Zielapplikation auf eine spätere Wiederausführung vor und entfernt vom Debugger hinterlassene Spuren. Der `pc` wird dabei auf die Adresse der Trap-Instruktion zurückgesetzt, die ihrerseits durch das Original-Datum ersetzt wird. Diese Entfernung des Breakpoints ist nötig, um die vorher durch den Trap-Befehl ersetzte Instruktion ausführen zu können.

Die Funktion `wait_for_inferior` hat die Aufgabe, auf den Stopp der Applikation zu warten und dessen Ursache, die vom Betriebssystem (UNIX) über den Status-Parameter des `wait`-Rufes zurückgegeben wird, auszuwerten. Ist die Stopp-Ursache das Erreichen eines Breakpoints, so werden alle vom Benutzer spezifizierten Breakpoints mit dem derzeitigen `pc` des Prozesses verglichen. Wird kein Thread-Debugging betrieben, so kehrt `wait_for_inferior` beim ersten Auftreten eines passenden Breakpoints zurück. Ist der Breakpoint jedoch an einen Thread geknüpft, so muß der Thread ermittelt werden, der den Breakpoint erreicht hat. Stimmen der vom Nutzer angegebene und der derzeitige Thread überein, so kann `wait_for_inferior` ebenfalls zurückkehren, und die `proceed`-Funktion kann beendet werden. Damit wäre auch der die `proceed`-Funktion rufende Befehl beendet.

Beim Prozeß-Debugging wird bei jedem Erreichen eines Breakpoints die Kontrolle an den Nutzer übergeben. Beim Thread-Debugging kann es jedoch passieren, daß der derzeitige Thread einen Breakpoint erreicht hat, dieser jedoch für einen anderen Thread gesetzt wurde. Für diesen Fall ist es notwendig, Änderungen an der Funktion `wait_for_inferior` vorzunehmen. Stimmt der derzeitige Thread nicht mit dem angegebenen überein, so muß die Applikation weiter ausgeführt werden, bis der angegebene Thread den Breakpoint erreicht. Die Abbildung 8.1 zeigt, wie die Funktion `resume` zum wiederholten Ausführen der Applikation gerufen wird.

Um den `resume`-Ruf und damit die Weiterausführung der Applikation zu ermöglichen, müssen die in Abbildung 8.2 skizzierten Operationen ausgeführt werden, da die Funktion `normal_stop`, die diese Funktionalität bereits hat, erst nach Beendigung von `wait_for_inferior` gerufen wird.

Um den Lauf der Applikation fortzusetzen, muß die durch die Trap-Instruktion maskierte Instruktion A ausgeführt werden, da sonst die Semantik des Programmes verändert würde. Nach der Ausführung der Instruktion A muß an deren Adresse $a[n]$ wieder der alte Breakpoint gesetzt werden. Wie in Abbildung 8.2 dargestellt, wird zuerst die Instruktion A eingefügt und der `pc` auf $a[n]$ gesetzt. Um später die Möglichkeit zu haben, den Breakpoint bei $a[n]$ wieder einzufügen, muß nach der Ausführung von A an der Stelle $a[n+1]$ gestoppt werden. Ein entsprechender temporärer Breakpoint wird eingefügt. Nach der Ausführung von A im zweiten Schritt stoppt die Applikation. Der Debugger fügt nun B wieder ein und setzt den `pc` auf $a[n+1]$ und den Breakpoint bei $a[n]$. Nun kann der Applikationslauf fortgeführt werden. Erlaubt die Hardware ein single-step, das heißt die Ausführung einzelner Instruktionen mit anschließendem Stopp, so kann das Einfügen des Breakpoints bei B entfallen. Die Instruktion A wird dann im single-step ausgeführt, sodaß der Breakpoint bei $a[n]$ wieder eingesetzt werden kann.

Hinzuzufügen ist, daß während des in der Abbildung 8.2 skizzierten zweiten Schrittes, in dem die durch den Breakpoint maskierte Instruktion A ausgeführt wird, kein Scheduling passieren darf, wenn die Architektur ein single-step erlaubt. Da dann weder bei Instruktion A noch bei B ein Breakpoint gesetzt ist, würden unge-

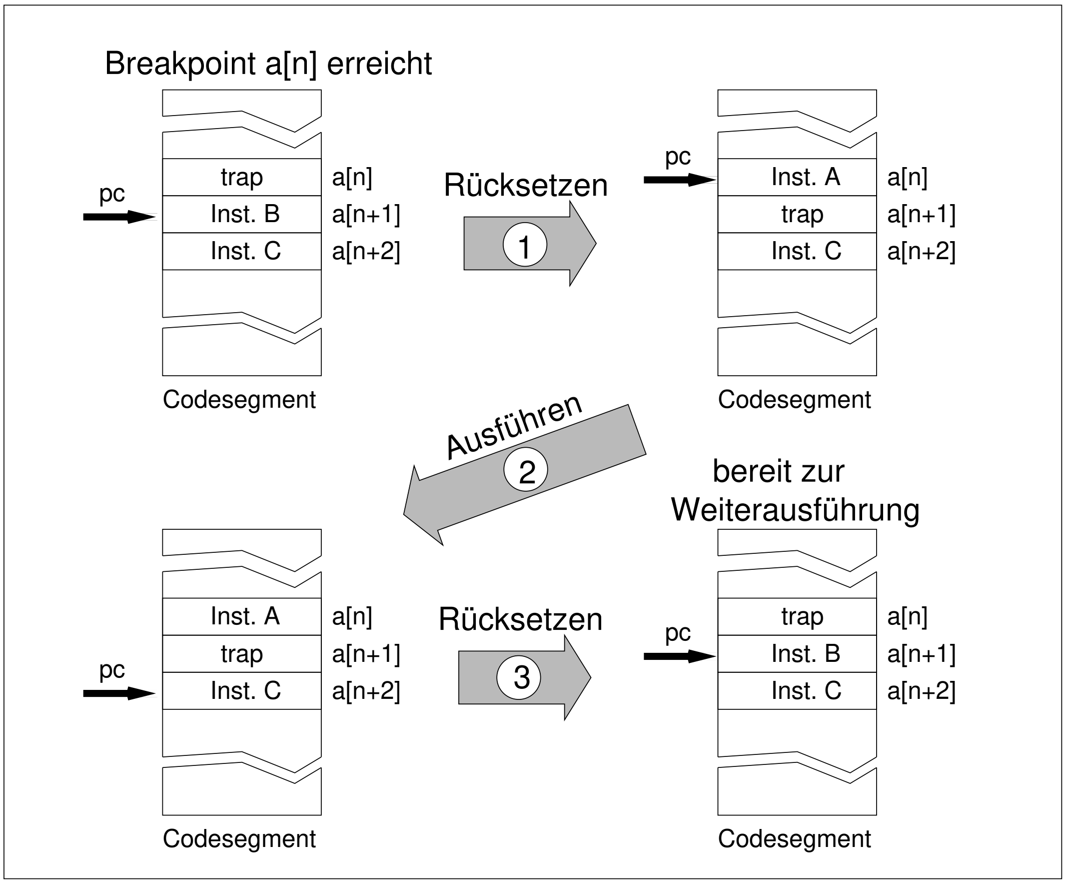

Abbildung 8.2.: Zurücksetzen eines Breakpoints

wollt geschedulte Threads diesen Breakpoint „verpassen". Deshalb wird während der Ausführung des zweiten Schrittes das Scheduling wie in Abschnitt 7.1.1.1 beschrieben deaktiviert.

Die eigentliche Benutzung des TDI bei der Umsetzung des threadspezifischen `break`-Befehls erfolgt bei der Ermittlung des Threads, der den Breakpoint auslöste. Der entsprechende TDI-Ruf für Uniprozessoren lautet:

```
...
TID_String = TDI.request("<TED>thread: id : state==<RUNNING>")
...
```

Bei Uniprozessoren kann nur der derzeit aktive Thread den Breakpoint ausgelöst haben. Bei der Verwendung von Kernel-Thread-Implementierungen auf Multiprozessoren kann diese Anfrage nicht benutzt werden, da hier mehrere Threads den Zustand `RUNNING` haben können.

Kernel-Thread-Implementierungen

Die fehlende Anfragemöglichkeit für Kernel-Threads ist nur auf den ersten Blick eine Einschränkung. Die TDI-Benutzung ist für Kernel-Thread-Implementierungen unnötig, da hier der auslösende Kernel-Thread für den Debugger sichtbar im User-Space vorhanden ist. Bei den Linuxthreads kann der auslösende Thread ohne TDI-Nutzung ermittelt werden, da die ID des dem POSIX-Thread zugeordneten Prozesses vom Systemruf `wait`, auf dem der Debugger für die Ausführung der Applikation blockiert, zurückgegeben wird.

Mischimplementierungen

Bei der Pthreads-Implementierung unter Solaris müssen sowohl TDI- als auch Systemfunktionen benutzt werden, um den auslösenden Thread zu ermitteln, da die Solaris-Implementierung der Pthreads eine Mischform (siehe Abschnitt 4.1.2.3) darstellt. Mit Hilfe des Debuggers ist es möglich, den Kernel-Thread zu ermitteln, der den Breakpoint ausgelöst hat. Detaillierte LWP-Informationen werden unter Solaris im `/proc`-Dateisystem zur Verfügung gestellt. Auf Multiprozessorsystemen kann ein Breakpoint von mehreren Threads parallel ausgeführt werden. Für diesen Fall muß sichergestellt sein, daß der Debugger seriell notifiziert wird, um den jeweils auslösenden LWP zu ermitteln. Um den dem LWP zugeordneten POSIX-Thread zu ermitteln, der über die Gültigkeit des Breakpoints entscheidet, wird das TDI benutzt. Die entsprechende Anfrage könnte lauten:

```
...
TID_String = TDI.Request("<TED>thread: id : lwp==<StopLWP>")
...
```

Das Thread-Attribut `lwp` enthält die LWP-ID des Kernel-Threads, von dem der POSIX-Thread zur Zeit ausgeführt wird. Diese Information muß in der Pthreads-Implementierung vorhanden sein.

Zusammenfassung

Da der GDB in der Lage ist, die Art der Pthreads-Implementierung festzustellen, kann er den auslösenden Thread entsprechend durch Benutzung des TDI bei User-Space-Implementierungen, durch Auswerten der von `wait` zurückgegebenen Prozeß-ID bei Kernel-Thread-Lösungen oder im Fall der gemischten Implementierungsform durch den kombinierten `/proc`- und TDI-Zugriff ermitteln.

8.4.2. Threadbindung der Einzelschritt-Kommandos

Die GDB-Befehle **step** und **next** ermöglichen es, die Applikation zeilenweise auszuführen, wobei **step** in gerufene Funktionen springt und **next** einen Funktionsaufruf als einzelnen Befehl betrachtet. Die Befehle **stepi** und **nexti** führen jeweils nur einzelne Instruktionen aus und haben in Bezug auf Funktionsrufe das Verhalten der Befehle **step** respektive **next**. Alle genannten Befehle erlauben es, wiederholt ausgeführt zu werden. Hierzu kann der Nutzer einen C-Ausdruck angeben, dessen Wert die Wiederholungsanzahl angibt, im einfachsten Fall eine natürliche Zahl.

Die Modifikation des GDB im Hinblick auf die Threadbindung der Befehle konnte bedingt durch die vorhandene umfangreiche Funktionalität des GDB schnell realisiert werden. Der GDB beinhaltet die universell nutzbare Funktion `step_1`, welche folgenden Prototyp besitzt:

```
static void
step_1 (int skip_subroutines, int single_inst, char* count_string)
```

Der boolesche Parameter `skip_subroutines` entscheidet über die Art der Funktionsbehandlung und damit, ob eine `step`- oder eine `next`-Semantik eingehalten werden soll. `single_inst` gibt an, ob Zeilen oder Einzelinstruktionen betrachtet werden und `count_string` ist die Zeichenkette des C-Ausdrucks, dessen Wert die Wiederholungsanzahl angibt.

Die Funktion `step_1` hat vereinfacht folgende Struktur:

```
step_1 ( ... , ... , ... )
{
 count = evalExp(count_string)

 for ( ; count > 0; count--)
 {
   step_internal(skip_subroutines,single_inst)
 }

}
```

Die Funktion `step_internal` hat die Aufgabe, bei der folgenden Zeile respektive Instruktion des Kontrollflusses zu stoppen. Bezogen auf die Threadbindung der Funktion `step_internal` muß ihre Aufgabe neu definiert werden. Eine threadgebundene Funktion `step_internal` hat die Aufgabe, bei der folgenden Zeile respektive Instruktion des Threads zu stoppen, der bei Ausführungsbeginn des Befehls aktiv war. Führt man die herkömmliche Funktion `step_internal` solange aus, bis der Ausgangs-Thread erreicht ist, erfüllt man die Kriterien eines threadgebundenen `step_internal`. Die Funktion `step_1` wurde wie folgt modifiziert, um eine Threadbindung zu erreichen:

```
step_1 ( ... , ... , ..., char* ThreadID )
{
 count = evalExp(count_string)

 for ( ; count > 0; )
```

```
 {
   step_internal(skip_subroutines,single_inst)

   if (equal_string(ThreadID,
                    TDI.request("<TED>thread:id:state==<RUNNING>")))
     count--;
 }

}
```

Das Dekrementieren der Variable `count`, das bei einem erfolgreich ausgeführten `step_internal` vollzogen wird, geschieht entsprechend nur, wenn die nächste Zeile oder Instruktion im Kontrollfluß des spezifizierten Threads erreicht wird.

Die threadspezifischen GDB-Kommandos zur Programmausführung erhielten das Präfix “`pt`” und haben die beschriebene Semantik, sodaß nun der Befehlssatz **ptnext**, **ptnexti**, **ptstep** und **ptstepi** zur Verfügung steht.

8.5. Globale Bindung

Die globale Threadbindung soll es ermöglichen, ausschließlich den Kontrollfluß **eines** Threads zu verfolgen. Für die globale Bindung der GDB-Kommandos an einen Thread wurden die Befehle

- **ptattach** <*Thread-ID*> und
- **ptdetach** eingeführt.

ptattach stoppt den angegebenen Thread T und bindet danach alle Befehle an diesen Kontrollfluß. Das heißt, daß alle Standard-Kommandos des GDB benutzt werden können, diese aber nach einem **ptattach** die Semantik der threadspezifischen Kommandos aus Abschnitt 8.4 haben, wobei alle Kommandos an den Thread T gebunden werden. Bei einem **ptattach** werden alle vorher gesetzten Breakpoints ausgeblendet, die nicht an den Thread T gebunden sind.

Mit dem Kommando **ptdetach** wird die Kontrollflußbindung der GDB-Befehle aufgehoben. Ihre Anwendung erfolgt danach wieder im Prozeß-Kontext. Alle ausgeblendeten Breakpoints werden deaktiviert.

8.6. Scheduling

Ein Thread-Debugger sollte dem Nutzer die Möglichkeit bieten, das Scheduling zu beobachten und Veränderungen vorzunehmen. Die folgenden Abschnitte beschreiben die Realisierung entsprechender GDB-Befehle.

8.6.1. Beobachtung von Kontextwechseln

Um die Arbeitsweise des Schedulers zu beobachten, ist es sinnvoll, die Kontextwechsel zu betrachten. Von besonderem Interesse sind dabei der Zeitpunkt des nächsten Kontextwechsels und die daran beteiligten Threads. Dem **continue**-Kommando des GDB wurde zu diesem Zweck die Option -**cs** (für ,,context switch") hinzugefügt. Ein **continue** -**cs** bewirkt die Programmausführung bis zum nächsten Kontextwechsel und die Ausgabe der ID des neuen Threads.

Der Befehl **continue** -**cs** wurde wie folgt umgesetzt. Gegeben sei die in Abbildung 8.3 dargestellte Ausgangssituation. Thread T_1 hat den Breakpoint B_1 erreicht; der Prozeß ist gestoppt. Da dem Debugger die Ausführungsreihenfolge der Threads nicht bekannt ist, müssen überall dort Breakpoints gesetzt werden, wo andere Threads ihre Ausführung fortsetzen könnten. Somit wird erreicht, daß die Applikation nach einem Kontextwechsel stoppt. Der Wert des Thread-Attributes `newpc`, dessen Ermittlung in Abschnitt 7.2.1.2 beschrieben wurde, entspricht der Programm-Adresse, bei der ein derzeit nicht aktiver[2] Thread seine Ausführung fortsetzt. Der erste Schritt des Befehls **continue** -**cs** besteht nun darin, für die

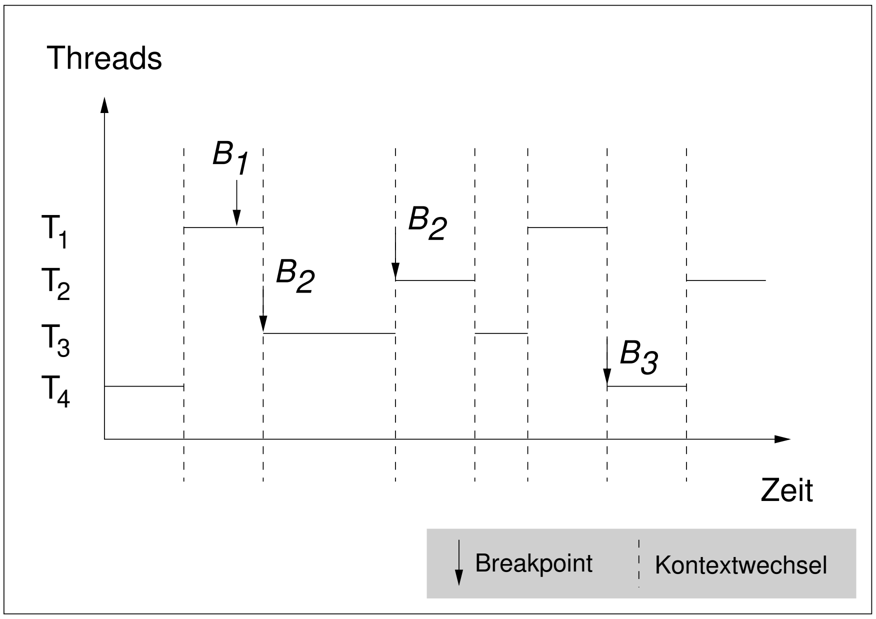

Abbildung 8.3.: Zurücksetzen eines Breakpoints

Menge der ermittelten Werte von `newpc` Breakpoints zu setzen. Wie in Abbildung 8.3 dargestellt, ist es möglich, daß zwei Threads (T_2,T_3) ihre Ausführung an gleicher Stelle fortsetzen, sodaß hier nur ein Breakpoint (B_2) gesetzt werden muß. Nach dem Setzen der Breakpoints kann die Applikation mit **continue** angestoßen

[2]Für einen aktiven Thread ist der Wert von `newpc` nicht ermittelbar, da im TED nicht vorhergesehen werden kann, wann dieser Thread gescheduled wird.

werden. Stoppt die Applikation, so wird der aktuelle Thread ausgegeben und alle `newpc`-Breakpoints werden entfernt.

8.6.2. Eingriff in das Scheduling

Ein Thread-Debugger für POSIX-Threads-Anwendungen sollte Möglichkeiten bieten, den Programmzustand auf Pthreads-Abstraktionsniveau zu ändern. In Bezug auf das Scheduling wurden hier die folgenden Operationen umgesetzt:

ptsuspend	*<Thread-ID-Liste>*	-	Aussetzen der Threadausführung,
ptresume	*<Thread-ID-Liste>*	-	Wiederaufnahme der Ausführung von mit **ptsuspend** suspendierten Threads und
ptswitch	*<Thread-ID>*	-	Vergabe der CPU an den spezifierten Thread und Fortführung der Applikation.

Unter Benutzung des TDI lassen sich diese Befehle leicht realisieren. In Abschnitt 6.2.1 wurde bereits erläutert, wie mit Hilfe der Änderung des Thread-Attributes `state` Einfluß auf das Scheduling genommen wird. Der größte Realisierungsaufwand für die Umsetzung der drei Operationen verlagert sich damit auf die Entwicklung der von der Pthreads-Implementierung im Rahmen des TED bereitgestellten Schreibfunktion für das Attribut `state`.

Die Kommandos **ptsuspend** und **ptresume** sind sehr ähnlich, da sie lediglich eine TDI-Anfrage zur Änderung des Attributes `state` durchführen. Im folgenden ist die Implementierung der Kommandos **ptsuspend** und **ptresume** skizziert. Da sich die Funktionen zur Umsetzung der Kommandos nur in einem Punkt unterscheiden, ist hier eine universelle über den Parameter `mode` steuerbare Funktion notiert.

```
#include <tdi.h> /* Definition der State-Kodierung */

...

void
ptsuspend_resume_command( ..., char *threadIDList, ModeT mode)
{
  ...

  /* Zusammensetzen des Selektionsausdruckes */

  FOR_EACH (thread,threadIDList)
   {
     IF_FIRST { strcat(selection,"id==%d",atoi(thread)); continue; }
```

```
      strcat(selection," || id==%d",atoi(thread));
    }

  /* Anfrage bilden ...  */

  sprintf(request,"<TED>thread:state=%d:%s",
                  (mode==SUSPEND_MODE)?
                   TDI_BLOCKED_S_STATE: /* suspend */
                   TDI_READY_STATE,     /* resume  */
                  selection)

  /* ... und absetzen.   */

  response = TDI.request(request)
  ...
}
```

Der Vorteil des TDI, das eine deklarative Anfragemöglichkeit bietet, wird bei der Implementierung von Befehlen, die auf Listen arbeiten, besonders deutlich. Es ist lediglich eine TDI-Anfrage nötig, um den Zustand aller in der Liste spezifizierten Threads zu ändern. Die Alternative bestünde darin, die Schreibfunktion des Attributes `state` für jeden Thread zu rufen, was einen erheblichen Mehraufwand für das wiederholte Vorbereiten des Funktionsaufrufes bedeutete. Im *SmartGDB* [Hal96] wird dieser ineffiziente Ansatz konzeptbedingt verfolgt, da keine Abarbeitung deklarativer Anfragen in der Laufzeitumgebung des Zielprozesses erfolgt.

Der Befehl **ptswitch** ist nur auf **einen** Thread anwendbar und erfordert zusätzlich zum Schreiben des Attributes `state` mit dem Wert `RUNNING` eine nachträgliche Signalisierung der Applikation. Dies ist damit begründet, daß während der Ausführung der TDI-Funktionen in der Laufzeitumgebung der Applikation kein Scheduling stattfinden darf. Das Setzen des Attributes `state` auf den Wert `RUNNING` bewirkt, daß der spezifizierte Thread an die erste Position der Warteschlange der ausführungsbereiten Threads gestellt wird. Um dem Thread die CPU zuzuteilen, wird der Applikation ein Signal gesendet, dessen Behandlung nach der Wiederausführung der Applikation erfolgt und ein Scheduling des Threads bewirkt.

Bei **ptsuspend** und **ptresume** werden lediglich Umordnungen in den Warteschlangen vorgenommen, keiner der spezifierten Threads erhält CPU-Zeit. Sollte aufgrund der Umordnungsoperationen ein Anstoßen des Schedulings notwendig sein, so kann dies durch die Pthreads-Laufzeitumgebung signalisiert werden. Im Abschnitt 7.1.1.1 wurde der Mechanismus für das Abschalten der Signalbehandlung während der Ausführung des TDI beschrieben. Die zu diesem Zweck in der Pthreads-Implementierung zusätzlich definierte Variable `pthread_debug_TDI_ignored_signals` kann benutzt werden, um ein eventuell nötiges Signalisieren der Applikation durch den Debugger zu erzwingen. Das Bit des für das Scheduling nötigen Signals wird gesetzt. Der Debugger interpretiert dieses Signal als ignoriert und sendet es nach der TDI-Ausführung erneut an die Applikation.

Im Zusammenhang mit dem Setzen des Attributes `state` ist hinzuzufügen, daß nicht bei allen vom Nutzer spezifierten Threads ein Umsetzen des Wertes möglich ist. Insbesondere ist ein **ptresume** nur bei Threads möglich, auf die ein **ptsuspend** angewendet wurde.

8.7. Zusammenfassung

Mit der Bereitstellung des TDI wird ebenfalls ein modifizierter GDB zur Verfügung gestellt, um die Möglichkeiten des TDI zu demonstrieren. Neben einigen Änderungen an den Quellen des GDB zur Umsetzung threadspezifischer Breakpoints (**break ... pthread**) und den Funktionen zur Fortführung des Ziel-Programmes bis zum nächsten Stopp im spezifizierten Kontrollfluß (**stepi**, **nexti**, **step**, **next**) wurden alle neuen Befehle in der Datei `pthread.c` zusammengefaßt. Die Befehle **ptattach**, **ptdetach**, **info pthreads**, **info pmutex**, **info pcond**, **tdi**, **ptsuspend**, **ptresume** und **continue -cs** sind hier implementiert. Neue Versionen können unter der WWW-Adresse `http://www.informatik.hu-berlin.de/~mueller/TDI` bezogen werden.

9. Schlußbetrachtungen

9.1. Ergebnis

Im Rahmen dieser Arbeit wurde eine einheitliche Thread-Debug-Schnittstelle (TDI) geschaffen, die es ermöglicht, Pthreads-Anwendungen unabhängig von der Art der verwendeten POSIX-Thread-Implementierung zu debuggen. Ein für das TDI angepaßter Debugger kann somit für alle POSIX-Threads-Applikationen verwendetet werden. Bisherige Thread-Debugger-Lösungen sind zumeist nur für eine Thread-Implementierung geeignet (KDB [BKS96], MpD [PHK91], Mach Debugger [CB90]).

Die im POSIX-Standard definierten Objekte und deren Beziehungen können mit Hilfe des TDI dargestellt werden. Das TDI bietet einen anfrageorientierten Zugang zur Pthreads-Laufzeitumgebung. Dieser einfache nachrichtenbasierte Zugang ermöglicht die schnelle und unkomplizierte Anpassung eines beliebigen Breakpoint-Debuggers im Hinblick auf ein Pthreads-Debugging.

Die Unterteilung des TDI in einen generischen Kern und eine implementierungsspezifische TED-Schicht ermöglicht eine leichte Erweiterbarkeit. Der Vorteil der Verwendung des TDI besteht darin, daß bei der Erweiterung der TDI-Debug-Unterstützung auf eine bisher nicht unterstützte Pthreads-Implementierung keine Änderungen des TDI-Kerns oder des Debuggers notwendig sind. Der Pthreads-Implementierung muß lediglich die TED-Funktionalität hinzugefügt werden. Der *SmartGDB* [Hal96] untertützt ebenfalls mehrere Pthreads-Implementierungen. Die Erweiterung seiner Debug-Unterstützung ist jedoch mit einer Reimplementierung verbunden.

Durch die Definition der Schnittstellen des TDI zu den Anrainern Debugger und Pthreads-Implementierung ist eine von diesen Komponenten unabhängige Entwicklung des TDI-Kerns möglich. Im Gegensatz dazu ist der *SmartGDB*, der als einziger existierender Thread-Debugger mehrerere Implementierungen der Pthreads unterstützt, abhängig von Änderungen der Pthreads-Implementierung, da seine Debug-Unterstützung auf dem Auslesen interner implementierungsspezifischer Daten festen Typs beruht.

Die erarbeitete Lösung ist anwendbar für Kernel- und User-Space-Implementierungen der POSIX-Threads. Exemplarisch wurden die MIT Pthreads [PHR93], die

FSU Pthreads [Mue93] sowie die Linuxthreads [Ler96] angepaßt. Die Anwendung auf Einprozessorsystemen wurde gezeigt und erscheint für Mehrprozessorsysteme möglich.

9.2. Ausblick

Diese Arbeit soll als Basis für Weiterentwicklungen des TDI verstanden werden. Es wurden Probleme und Lösungsansätze erarbeitet, die für eine Reihe von Implementierungsformen der Pthreads gelten. Die offenen Punkte, die im Rahmen dieser Arbeit nicht mehr behandelt werden konnten, und die Vorschläge für die künftige Entwicklung des TDI sollen im folgenden angesprochen werden.

9.2.1. Ineffizienz threadspezifischer Beakpoints

Die Benutzung von Breakpoint-Debuggern für das Thread-Debugging erfordert die Bereitstellung von Mitteln für das Setzen threadspezifischer Breakpoints.

9.2.1.1. Konventionelle Lösung

In der hier vorgestellten Modifikation des Gdb gibt es keinen Unterschied zwischen dem Setzen eines prozeß- und eines threadspezifischen Breakpoints. Bei threadspezifischen Breakpoints wird bei jedem Erreichen der Trap-Instruktion das Programm gestoppt. Danach wird mit Hilfe des TDI der aktuelle Thread ermittelt. Stimmt dieser mit dem Thread überein, an den der Breakpoint gebunden ist, so wird dem Nutzer des Debuggers die Kontrolle übergeben. Anderenfalls wird die Ausführung des Zielprozesses fortgesetzt. In Abbildung 9.1 ist dieses Szenario skizziert. Der Nachteil dieser Lösung ist, daß die Applikation auch dann gestoppt wird, wenn ein nicht spezifizierter Thread stoppt. Dies führt aufgrund der erforderlichen Prozedur für das Rücksetzen und Setzen des Breakpoints (siehe Abschnitt 8.4.1) zu einem hohen Laufzeitverbrauch. Bei User-Space-Implementierungen muß der auslösende Thread mit Hilfe des TDI ermittelt werden. Dazu ist die Simulation eines Funktionsrufes in der Zielapplikation nötig, was einen zusätzlichen Aufwand bedeutet.

9.2.1.2. Fast Breakpoints

In der Arbeit von Kessler [Kes90] wird eine Methode zur effizienten Realisierung threadspezifischer Breakpoints vorgestellt. Der verfolgte Ansatz verlagert den Vergleich zwischen dem spezifizierten und dem auslösenden Thread in die Applikation. Der konventionelle Ansatz bei der Realisierung von Breakpoints ist die Ersetzung der ursprünglichen Instruktion durch eine Trap-Instruktion. Bei Ausführung dieser

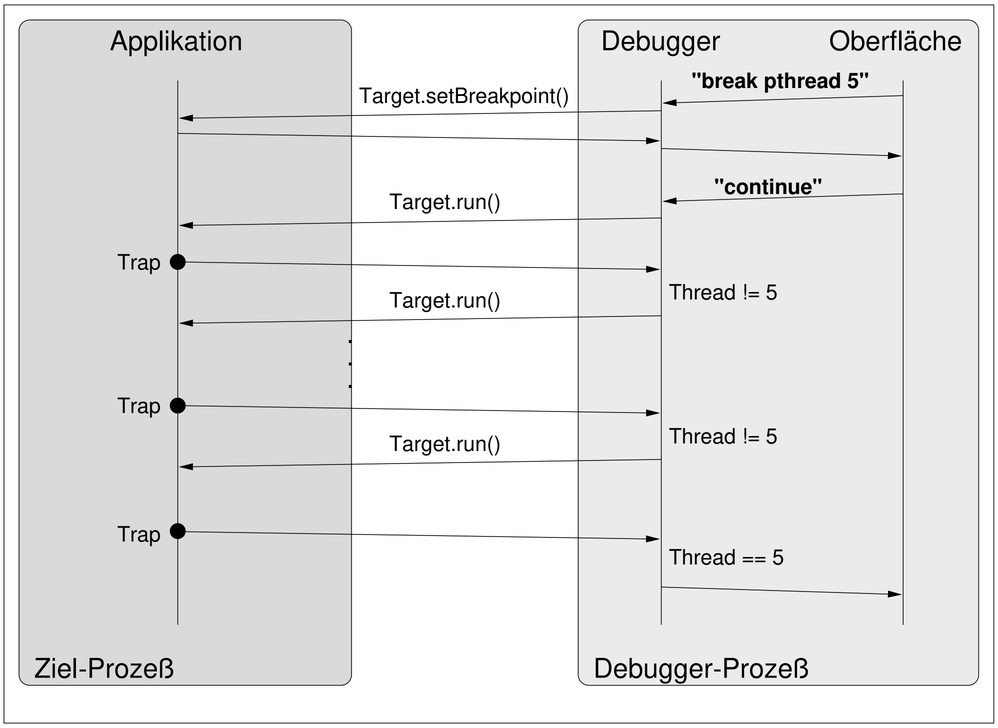

Abbildung 9.1.: Derzeitige Realisierung threadspezifischer Breakpoints

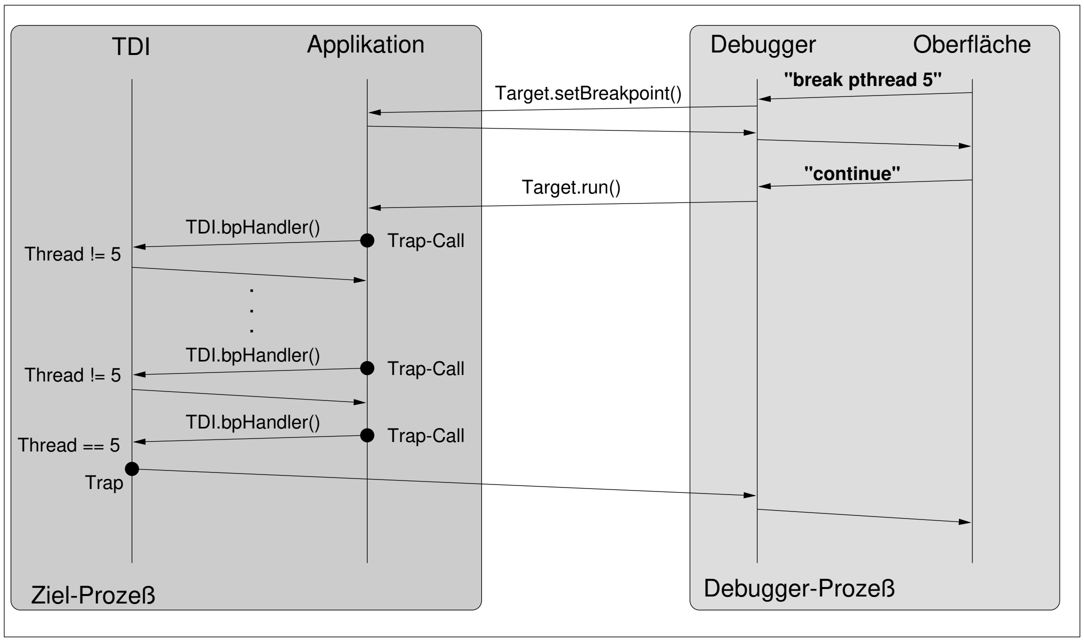

Abbildung 9.2.: Effiziente Lösung mit *fast breakpoints*

Instruktion wird ein Software-Interrupt ausgelöst und der Debugger wird notifiziert.

Zur Realisierung von threadspezifischen Breakpoints wurde der konventionelle Ansatz durch Kessler abgewandelt. Er ersetzt die Instruktion I_{old}, bei der der Breakpoint gesetzt werden soll, durch den Aufruf einer Vergleichsfunktion. Diese Funktion vergleicht den derzeitigen Thread mit dem spezifizierten. Stimmen diese nicht überein, so wird die Instruktion I_{old} ausgeführt, um die Programmsemantik zu erhalten. Sind die Threads identisch, wird die Trap-Instruktion ausgeführt und der Debugger wird benachrichtigt. Abbildung 9.2 illustriert dieses Verfahren. Da alle Threadvergleiche im Ziel-Prozeß und ohne die Intervention des Debuggers ausgeführt werden, ist diese Lösung sehr effizient.

Für die Zukunft sollte eine Realisierung der „fast breakpoints“ angestrebt werden. Das TDI könnte dabei eine wichtige Rolle spielen. Da der Debugger die Breakpoint-Instruktion durch einen Aufruf der Vergleichsfunktion ersetzen muß, wäre es sinnvoll, eine einheitliche Funktion zu definieren, um eine von der jeweiligen Pthreads-Lösung unabhängige Implementierung des Debuggers zu ermöglichen. Diese definierte Vergleichsfunktion könnte im TDI-Kern plaziert werden (siehe Abbildung 9.2) und wäre dem Debugger bekannt.

9.2.2. Prä- und Postprozessor für direkte TDI-Anfragen

Der hinzugefügte GDB-Befehl **tdi** ermöglicht es dem Nutzer, TDI-QL-Anfragen zu stellen. Diese Anfragemöglichkeit ist derzeit noch sehr unkomfortabel.

Alle Anfragen müssen so gestellt werden, wie sie vom Parser der TDI-Anfragebearbeitung bearbeitet werden. Alle Konstanten für die Attributwerte müssen derzeit dezimal oder hexadezimal angegeben werden. In Zukunft sollte ein Präprozessor die vom Nutzer spezifizierzten Anfragen in eine für das TDI geeignete Form umwandeln. So sollte es dann möglich sein, Symbolnamen anstelle ihrer Werte zu benutzen. In der Nutzeranfrage

```
,,<TED>thread:id,prio:state==RUNNING && entry==main && prio<max_prio-10''
```

könnten für `RUNNING`, `main` und `max_prio` die entsprechenden Werte eingesetzt werden. `RUNNING` wird durch den im TDI definierten Ganzzahlwert des Zustandes, `main` durch die Adresse der Funktion und `max_prio` durch den Wert der in der Applikation definierten Variable ersetzt.

Die Ausgaben des **tdi**-Kommandos entsprechen dem in Abschnitt 6.3.3 definierten und für den Nutzer umständlich lesbaren Format. Um die Lesbarkeit der Ausgaben zu erhöhen, sollte ein Postprozessor eingesetzt werden. In der Anfragezeichenkette sind die ausgegebenen Attribute und deren Reihenfolge kodiert, sodaß diese Informationen vom Postprozessor benutzt werden können, um eine Formatierung und Konstantenumwandlung vorzunehmen. Die Formate der Attribute (hexadezimal, dezimal, Symbolname usw.) könnten konfigurierbar sein.

Prä- und Postprozessor wären weitesgehend TDI-unabhängig und nutzten lediglich die Konstantendefinition des TDI (`tdi.h`).

9.2.3. Einschränkungen

Nach der Veröffentlichung des TDI in dem für die Thread-Programmierung relevanten Diskussionsforum `comp.programming.threads` wurden erste Fragen durch die Diskussionsteilnehmer aufgeworfen. Dave Butenhof, Autor der Buchreihe „Programming with Threads“ [But97], machte folgende Bemerkungen zum TDI, die im folgenden kurz diskutiert werden sollen. Folgende Probleme wurden angesprochen:

i) Ist ein post-mortem-Debugging mit Hilfe des TDI möglich?

ii) Besteht die Gefahr der Änderung des Applikationsverhaltens bei Einsatz des TDI?

iii) Ist es möglich, daß die Funktionsweise des TDI durch ein fehlerhaftes Programmverhalten beeinträchtigt wird (z.B. durch Überschreiben vom TDI gehaltenen Registrierungsdaten)?

Ein post-mortem-Debugging (**i)**), das heißt das Debugging mit Hilfe des gespeicherten Prozeßabbildes einer Applikation, ist mit dem TDI möglich. Die Vollständigkeit der Informationen ist jedoch davon abhängig, ob die Applikation mit Hilfe des TDI debugged wurde. Wurde das TDI bereits vor dem Prozeßende benutzt, so sind alle beim TDI registrierten Objekte noch im Speicherabbild des Prozesses vorhanden und können mit den im TED bereitgestellten Zugriffsfunktionen auch post-mortem gelesen werden. Die Objekte, deren Menge von der Pthreads-Laufzeitumgebung verwaltet werden, sind auch ohne die vorherige Verwendung des TDI post-mortem sichtbar, wenn die Iteratoren des TED auf diesen internen Datenstrukturen der Pthreads-Implementierung arbeiten.

Zu **ii)** ist zu bemerken, daß der Einsatz des TDI eine Änderung des Applikationsverhaltens bewirken kann, da der Prozeß während der TDI-Ausführung CPU-Zeit erhält. Einige Maßnahmen zur Prävention eines geänderten Applikationsverhaltens sind in Abschnitt 7.1.1 beschrieben. Weiterhin ist anzumerken, daß der Ausführungsstack des TDI auf dem Stack des derzeit aktiven Threads der Applikation ausgeführt wird, wodurch sich unter Umständen Probleme bei gering bemessenen Stackgrößen ergeben. Potentielle Verhaltensänderungen sind bei User-Space-Implementierungen kaum vermeidbar, da alle Threads in einem Kernel-Thread ausgeführt werden. Bei Kernel-Thread-Implementierungen kann das TDI in einem separaten Thread und damit ohne die genannten Wechselwirkungen ausgeführt werden.

Dem in **iii)** angesprochenen Problem des unzulässigen Überschreibens von TDI-Daten durch die Applikation kann durch die Verwendung eines mit (`mprotect()`)

geschützten Speicherbereiches begegnet werden. Alle vom TED und TDI benutzten Daten könnten innerhalb dieses Datenbereiches angelegt werden. Diese Lösung ist portabel (POSIX-Standard) und könnte im Rahmen kommender Änderungen am TDI umgesetzt werden. Da `mprotect()` bei jeder TDI-Anfrage gerufen wird, ist diese Lösung jedoch sehr ineffizient.

9.2.4. Generelle Weiterentwicklung des TDI

Mit Fertigstellung dieser Arbeit wird das TDI im Sinne der GNU Library General Public License [FSF96] freigegeben. Mit der Freigabe des Quellcodes ist der Wunsch verbunden, die Gemeinde der Anwendungs- und Pthreads-Entwickler für die Weiterentwicklung des TDI zu gewinnen.

Die Akzeptanz des TDI ist im wesentlichen von der Qualität der TED-Implementierungen für die einzelnen Pthreads-Implementierungen und dem komfortablen Zugang seitens des Debuggers abhängig. Eine hohe Qualität kann erreicht werden, da sich die Zahl der Testpersonen, der Testplattformen sowie der verwendeten Pthreads- und Debugger-Implementierungen erhöht. Vorhandene Fehler und Unzulänglichkeiten können schneller entdeckt und dank der verteilten Kompetenz der Nutzer behoben werden.

Im Zuge der Publikation des TDI's wurden die folgenden öffentlichen Zugangspunkte geschaffen:

TDI-Quellpaket, Patches und weitere Informationen im WWW :
http://www.informatik.hu-berlin.de/∼mueller/TDI

E-Mail-Adresse für Rückmeldungen und den Fehlerreport :
tdi-bugs@informatik.hu-berlin.de

Literaturverzeichnis

[Ame89] AMERICAN NATIONAL STANDARDS INSTITUTE: *ANSI X3.135-1992: Information Systems — Database Language — SQL (includes ANSI X3.168-1989).* American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1989.

[BKS96] BUHR, PETER A., MARTIN KARSTEN und JUN SHIH: *KDB: A Multi-threaded Debugger for Multi-threaded Applications.* In: *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, Seiten 80–87, Philadelphia, Pennsylvania, USA, Mai 1996. ACM Press.

[Boo94] BOOCH, G.: *Object-Oriented Analysis and Design.* Benjamin Cummings, 1994.

[But97] BUTENHOF, DAVID R.: *Programming with POSIX threads.* Addison-Wesley, Reading, MA, USA, 1997.

[CB90] CASWELL, DEBORAH und DAVID BLACK: *Implementing a Mach Debugger for Multithreaded Applications.* In: USENIX ASSOCIATION (Herausgeber): *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA*, Seiten 25–40, Berkeley, CA, USA, Januar 1990. USENIX.

[Cha94] CHAMBERLAIN, STEVE: *Using ld: The GNU linker.* Cygnus Support, Januar 1994.

[Cod70] CODD, E. F.: *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM, 13(6):377–387, Juni 1970. Also published in/as: 'Readings in Database Systems', M. Stonebraker, Morgan-Kaufmann, 1988, pp. 5–15.

[DS90] DONNELLY, CHARLES und RICHARD STALLMAN: *Bison: the Yacc-compatible parser generator.* Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, Bison Version 1.12 Auflage, Dezember 1990.

[FSF96] FOUNDATION FREE SOFTWARE: *The GNU Library General Public License.* http://www.gnu.org/philosophy/why-not-lgpl.html, 1996.

[Gai86] GAIT, JASON: *A Probe Effect in Concurrent Programs.* Software Practice and Experience, 16(3):225–233, März 1986.

[GH93] GOLAN, MICHAEL und DAVID R. HANSON: *DUEL — A Very High-Level Debugging Language.* In: USENIX ASSOCIATION (Herausgeber): *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, Seiten 107–117, Berkeley, CA, USA, Winter 1993. USENIX.

[Hal96] HALBHAVI, SUDHIR: *Thread Debugger Implementation and Integration with the SmartGDB Debugging Paradigm.* University of Kansas, 1996. [Master Thesis].

[IEE95] IEEE: *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language).* IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[Kes90] KESSLER, PETER B.: *Fast breakpoints. Design and implementation.* ACM SIGPLAN Notices, 25(6):78–84, Juni 1990.

[Ler96] LEROY, XAVIER: *The LinuxThreads Library.* http://pauillac.inria.fr/∼xleroy/linuxthreads, 1996.

[LMB92] LEVINE, JOHN R., TONY MASON und DOUG BROWN: *Lex & Yacc.* O'Reilly, Sebastopol, 2 Auflage, 1992.

[LMKQ88] LEFFLER, S. J., M. K. MCKUSICK, M. J. KARELS und J. S. QUARTERMAN: *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1988.

[MH89] MCDOWELL, CHARLES E. und DAVID P. HELMBOLD: *Debugging Concurrent Programs.* ACM Computing Surveys, 21(4):593–622, Dezember 1989.

[Mue93] MUELLER, FRANK: *A Library Implementation of POSIX Threads under UNIX.* In: USENIX ASSOCIATION (Herausgeber): *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, Seiten 29–41, Berkeley, CA, USA, Winter 1993. USENIX.

[Ope97] OPEN GROUP: *The Single UNIX Specification, Version 2.* http://www.UNIX-systems.org/single_unix_specification_v2, 1997.

[PB93] Pomberger, G. und G. Blaschek: *Software Engineering. Prototyping und objektorientierte Software-Entwicklung.* Carl Hanser Verlag, Muenchen, 1993.

[PHK91] Ponamgi, M. Krish, Wenwey Hseush und Gail E. Kaiser: *Debugging Multithreaded Programs with MPD.* IEEE Software, 6(3):37–43, Mai 1991.

[PHR93] Provenzano, C., G. Hudson und K. Raeburn: *MIT Pthreads.* http://www.mit.edu/people/proven/pthreads.html, 1993.

[POSa] POSIX: *longjmp (3).* manual page.

[POSb] POSIX: *setjmp (3).* manual page.

[Sch98] Schulz, Daniel: *POSIX Threads Debugger.* Humboldt Universität zu Berlin, 1998. [Studienarbeit].

[SCP97] SCPI Consortium: *SCPI Standard.* http://www.scpiconsortium.org, 1997.

[Sta89] Stallman, R. M.: *GDB Manual (The GNU Source-Level Debugger).* Technischer Bericht Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, Januar 1989. Third Edition, GDB version 3.1.

[Sta91] Stallman, Richard M.: *Using and Porting GNU CC.* Free Software Foundation, Inc., 1.39 Auflage, Januar 1991.

[Sta95] Stallman, Richard: *The GNU Manifesto.* http://www.fsf.org/gnu/manifesto.html, 1995.

[Ste98] Stevens, W. Richard: *UNIX Network Programming, Interprocess Communications*, Band 2. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, Zweite Auflage, 1998.

[Suna] Sun: *libthread_db (3t).* manual page (Solaris).

[Sunb] Sun: *pthreads (3t).* manual page.

[Tan95] Tanenbaum, A. S.: *Distributed Operating Systems.* Prentice-Hall, Inc., 1995.

[UNI] UNIX SVr4: *ptrace (2).* manual page.

[Vol89] Volkman, Victor R.: *Public Domain Lex Is Fast And Flex-ible.* C Users Journal, 7(10):129–??, Oktober 1989.

[Vos94] VOSSEN, GOTTFRIED: *Datenmodelle, Datenbanksprachen und Datenbankmanagement-Systeme*. Addison-Wesley, Bonn, 2 Auflage, 1994.

A. Relationen und Attribute

Im folgenden sollen die Attribute und deren Wertebereiche der Relationen Thread, Mutex und Bedingungsvariable aufgelistet werden.

Die Wertebereichsdefinitionen geben an, wie das Anfrageergebnis ausgewertet werden muß. Im Anfrageergebnis sind hexadezimal kodierte Werte, die den Wertebereichsdefinition entsprechen müssen, und das ausgezeichnete Element „*NULL*“ enthalten.

Attributname	Beschreibung	Wertebereich
`id`	prozeßpersistente Thread-ID	$dom(id) = \mathbb{N} \cup \{\text{NULL}\}$
`addr`	Speicheradresse der Thread-Struktur	$dom(addr) = \mathbb{N} \cup \{\text{NULL}\}$
`prio`	Priorität des Threads	$dom(prio) = \mathbb{Z} \cup \{\text{NULL}\}$
`state`	Ausführungszustand des Threads	$dom(state) = \{$*TDI_UNDEF_STATE=0, TDI_RUNNING_STATE, TDI_READY_STATE, TDI_BLOCKED_M_STATE, TDI_BLOCKED_C_STATE, TDI_BLOCKED_T_STATE, TDI_BLOCKED_S_STATE, TDI_BLOCKED_O_STATE, TDI_EXITING_STATE* $\}$
`rstate`	implementierungsspezifischer Zustandswert	$dom(rstate) = \mathbb{Z} \cup \{\text{NULL}\}$
`entry`	Adresse der Startfunktion des Threads	$dom(entry) = \mathbb{N} \cup \{\text{NULL}\}$
`earg`	Adresse des Argumentes der Startfunktion	$dom(earg) = \mathbb{N} \cup \{\text{NULL}\}$
`newpc`	Wert des Programmzählers, bei dem der Thread seine Ausführung fortsetzt.	$dom(newpc) = \mathbb{N} \cup \{\text{NULL}\}$
`sp`	Stackpointer des dem Thread zugeordneten Ausführungsstacks.	$dom(sp) = \mathbb{N} \cup \{\text{NULL}\}$
`mbo`	Mutex, auf dem der Thread blockiert ist.	$dom(mbo) = \mathbb{N} \cup \{\text{NULL}\}$
`cvwf`	Bedingungsvariable, auf deren Signalisierung der Thread wartet.	$dom(cvwf) = \mathbb{N} \cup \{\text{NULL}\}$
`pid`	Id des Prozesses, von dem der Thread ausgeführt wird.	$dom(pid) = \mathbb{N} \cup \{\text{NULL}\}$

Tabelle A.1.: Thread-Attribute

Attributname	Beschreibung	Wertebereich
`id`	prozeßpersistente ID	$dom(id) = \mathbb{N} \cup \{\text{NULL}\}$
`addr`	Speicheradresse der Mutex-Struktur	$dom(addr) = \mathbb{N} \cup \{\text{NULL}\}$
`owner`	ID des Threads, der den Mutex besitzt.	$dom(owner) = \mathbb{N} \cup \{\text{NULL}\}$

Tabelle A.2.: Mutex-Attribute

Attributname	Beschreibung	Wertebereich
`id`	prozeßpersistente ID	$dom(id) = \mathbb{N} \cup \{\text{NULL}\}$
`addr`	Speicheradresse der Mutex-Struktur	$dom(addr) = \mathbb{N} \cup \{\text{NULL}\}$
`cmutex`	ID des Mutex, der während des Wartens auf die Bedingungsvariable freigegeben ist.	$dom(cmutex) = \mathbb{N} \cup \{\text{NULL}\}$

Tabelle A.3.: Mutex-Attribute

B. TDI-QL Grammatik

Die Grammatik besteht aus drei Teilen, die für unterschiedliche Zwecke genutzt werden:

- Präkompiler für syntaktische und semantische Analyse (`Query`)
- Auswertung des Selektionsprädikates für jedes Tupel (`RuntimeEvaluation`)
- Administrative Aufgaben (bisher `Init`)

TDI-QL

Der folgende Ausschnitt der YACC-Datei beschreibt die Grammatik:

```
Command: Init                Terminator /* Administration                 */
       | Query               Terminator /* PreCompiler                    */
       | RuntimeEvaluation Terminator /* per-Tupel-Interpreter (intern) */
       ;

Query:   TEDMODE DistinctMode EntityAccess ':' SpecExpression ;

/* administrativer Befehl */
Init:    INIT ;

/* alle Tupel sollen ausgegeben werden (vgl. SQL: "ALL")
DistinctMode:  '!' | /* empty */ ;

/* welche Relation ? */
EntityAccess: THREADREL
            | MUTEXREL
            | CONDREL
            ;

/* RA-Operatoren */
SpecExpression: VList ':' SpecExpr /* Projektion und Selektion */
```

```
                | VList            /* nur Projektion */
                ;

/* Projektionliste */
VList:  VAttr ','  VList
         | VAttr
         ;

SpecExpr: SpecExpr AND SpecExpr
          | SpecExpr OR SpecExpr
          | '(' SpecExpr ')'
          | CmpExpr
          ;

CmpExpr: TermExpr LESS        Const
         | TermExpr LESS_EQUAL  Const
         | TermExpr GRT         Const
         | TermExpr GRT_EQUAL   Const
         | TermExpr EQUAL       Const
         | TermExpr NOTEQUAL       Const
         ;

TermExpr: CAttr
          | Const
          | TermExpr ASTERISK  TermExpr
          | TermExpr '/'  TermExpr
          | TermExpr '-'  TermExpr
          | TermExpr '+'  TermExpr
          ;

/* ---------------------------------------------------------- */
/* RT-Ausdrücke werden für jedes Tupel der Relation ausgewertet */

/* Selektionsausdruck */
RuntimeEvaluation: '.' RTSpecExpr ;

/* Boolesche Terme des Selektions-Prädikates */

RTSpecExpr: RTSpecExpr AND RTSpecExpr
            | RTSpecExpr OR RTSpecExpr
            | '(' RTSpecExpr ')'
            | RTCmpExpr
            ;

RTCmpExpr: RTTermExpr LESS        RTConst
           | RTTermExpr LESS_EQUAL  RTConst
```

```
            | RTTermExpr GRT         RTConst
            | RTTermExpr GRT_EQUAL   RTConst
            | RTTermExpr EQUAL       RTConst
            | RTTermExpr NOTEQUAL    RTConst
;

/* Arithmetik im Prädikat */
RTTermExpr: RTPlaceHolder
            | RTConst
            | RTTermExpr ASTERISK  RTTermExpr
            | RTTermExpr '/'  RTTermExpr
            | RTTermExpr '-'  RTTermExpr
            | RTTermExpr '+'  RTTermExpr
            ;

/* Platzhalter der Form '$'<Attributindex> */
RTPlaceHolder: '$' INT_CONST ;

/* Konstante */
RTConst: Const ;

/* ------------------------------------------------------------ */

CAttr: Attr;

VAttr: Attr  '=' Const
       | Attr
       ;

Attr: IDENT;

Const: INT_CONST
       | HEX_CONST
       ;

Terminator: /* empty */
            | '\n'
            ;
```

Präzedenz[1] und Assoziativität[2] [LMB92] für die aritmetischen und logischen Operatoren sind wie folgt definiert, wobei die erste Zeile der niedrigsten Präzedenz-

[1]Die Präzedenz bestimmt die Ausführungsreihenfolge von Operatoren (beispielseise '*' vor '-').

[2]Die Assoziavitität definiert die Reihenfolge der Operatorausführung bei gleicher Präzedenz. Es wird zwischen linksassoziativ und rechtsassoziativ unterschieden, z.B. für a+b+c ist (a+b)+c links- und a+(b+c) rechtsassoziativ.

Stufe entspricht:

<table>
<tr><th>Operator</th><th>Assoziativität</th></tr>
<tr><td>'||'</td><td>left</td></tr>
<tr><td>'&&','.'</td><td>left</td></tr>
<tr><td>'<','>','<=','>=','=='</td><td>left</td></tr>
<tr><td>'+','-'</td><td>left</td></tr>
<tr><td>'*','/'</td><td>left</td></tr>
</table>

Tabelle B.1.: Präzedenz-Stufen der Operatoren

Tokendefinition

Die regulären Ausdrücke der Token sind wie folgt definiert:

<table>
<tr><th>Token</th><th>Regulärer Ausdruck</th></tr>
<tr><td>IDENT</td><td>[a-zA-Z]+</td></tr>
<tr><td>INT_CONST</td><td>([1-9][0-9]*|0)</td></tr>
<tr><td>HEX_CONST</td><td>0x[0-9a-fA-F]+</td></tr>
<tr><td>LOG_AND</td><td>"&&"</td></tr>
<tr><td>LOG_OR</td><td>"||"</td></tr>
<tr><td>LESS</td><td>"<"</td></tr>
<tr><td>LESS_EQUAL</td><td>"<="</td></tr>
<tr><td>GRT</td><td>">"</td></tr>
<tr><td>GRT_EQUAL</td><td>">="</td></tr>
<tr><td>EQUAL</td><td>"=="</td></tr>
<tr><td>ASTERISK</td><td>"*"</td></tr>
<tr><td>NOTEQUAL</td><td>"!="</td></tr>
<tr><td>TEDMODE</td><td>"<TED>"</td></tr>
<tr><td>INIT</td><td>Init|INIT|init</td></tr>
<tr><td>MUTEXREL</td><td>MUTEX|mutex</td></tr>
<tr><td>CONDREL</td><td>COND|cond</td></tr>
<tr><td>THREADREL</td><td>THREAD|thread</td></tr>
</table>

Tabelle B.2.: TDI-QL-Token

C. Beispielsitzung mit modifiziertem GDB

Im folgenden werden anhand einer Beispielsitzung die neuen Thread-Debug-Befehle des modifizierten Gdb erläutert. Zuvor wird das in der Beispielsitzung verwendete Programm `calc.c` beschrieben.

C.1. Das Programm: calc.c

Es soll ein System modelliert werden, in dem die Betriebsmittel (Recheneinheiten) beschränkt sind und eine große Anzahl von Nutzern (Rechenaufträge) vorhanden ist. Gegeben sei eine Menge von Recheneinheiten R. Jede dieser Recheneinheiten kann genau eine der folgenden Operationen ausführen:

- Multiplikation zweier ganzer Zahlen und
- Addition zweier ganzer Zahlen.

Damit unterteilt sich die Menge R in die Menge der multiplizierenden Einheiten M und die der addierenden Einheiten A.

Weiterhin sei eine Menge von Rechenaufträgen J gegeben. Jeder Rechenauftrag $j \in J$ soll genau einen arithmetischen Ausdruck berechnen, der die obengenannten Operationen enthalten darf. Jeder Rechenauftrag j ist bei der Evaluierung seines arithmetischen Ausdrucks auf die Hilfe von Recheneinheiten angewiesen. Ist beispielsweise der Ausdruck $e = a * b + c$ zu berechnen, so muß zunächst ein $m \in M$ gefunden werden, das $y = a * b$ berechnet und dann ein $a \in A$, das $z = y + c$ berechnet. Als Ganzzahlwert von e kann nun z zurückgegeben werden.

Außerdem seien folgende Einschränkungen gegeben. Jede Recheneinheit $r \in R$ kann zum Zeitpunkt t nur eine Rechenoperation ausführen. Die Anzahl der Recheneinheiten ist beschränkt. Sei op(j) die Anzahl der Rechenoperationen für $j \in J$, die im von j zu berechnenden Ausdruck vorkommen. Dann gilt insbesondere $\sum_{j \in J} op(j) \gg |R|$. Es ist also nicht möglich, ohne eine Aufteilung der Ressourcen die Rechenaufträge auszuführen. Diese Aufteilung wird erreicht, indem die

Nutzung einer Recheneinheit nur im gegenseitigen Ausschluß erfolgen kann. Die Rechenaufträge müssen synchronisiert werden.

Sind alle Recheneinheiten von Rechenaufträgen benutzt, so müssen die restlichen Rechenaufräge warten. Sobald eine Recheneinheit r mit der Ausführung der von ihr angebotenen Operation fertig ist und ein j genau die von r angebotene Operation ausführen möchte, so kann er diese benutzen, sofern nicht andere Wartende vor ihm sind.

Eine Recheneinheit $r \in R$ sei folgendermaßen modelliert. Jedes r wird durch folgende Daten repräsentiert, die in der Struktur `u_tupel` enthalten sind.

```
struct u_tupel {
   int nr;
   pthread_mutex_t lock;
   pthread_cond_t  signal;
   int x; /* Input 1  */
   int y; /* Input 2  */
   int z; /* Ergebnis */
   int status; /* bereit, regs gefuellt, rechnend, fertig  */
   int type;   /* Unittyp: {MUL,ADD} */
 } unit_tupel[UNITS];
```

Jede Recheneinheit hat eine Schlüsselnummer `nr`, zwei Eingaberegister `x`, `y` und ein Ausgaberegister `z`. Zur Synchronisation dienen ein Mutex mit dem Namen `lock` und die Bedingungsvariable `signal`. Der Typ der Recheneinheit wird in der Variable `type` gespeichert. Die verschiedenen Zustände, in der sich ein r befinden kann, werden in `status` festgehalten. Die Zustände sind:

- rechenbereit (`S_READY`)
- Register gefüllt (`S_FILLED`)
- Ergebnis berechnet (`S_STOPPED`).

Die Daten aller Recheneinheiten werden im Feld `unit_tupel` gespeichert.

Die Zustandsüberführungen, die vollzogen werden, sind folgendermaßen definiert. Eine Recheneinheit befindet sich nach dem Start im Zustand `S_READY`. Will ein Job j diese Recheneinheit benutzen, so muß er sich den Mutex für diese Einheit sichern. Das geschieht durch ein `pthread_mutex_lock(`$r.lock$`)`. Jetzt kann er die Register `x`, `y` füllen. Danach setzt er den Status von r auf `S_FILLED`. In diesem Zustand ist die Recheneinheit für keinen Job mehr benutzbar. Die Recheneinheit wird nun durch den Rechenauftrag „aufgeweckt", indem ein `pthread_cond_signal(`$r.signal$`,`$r.lock$`)` ausgeführt wird. Die Recheneinheit r, die sich auf die Bedingungsvariable *r.signal* schlafengelegt hatte, kommt nun wieder in den Besitz des Mutex *r.lock* [1]. Nun führt die Recheneinheit ihre Operation auf den Eingaberegistern aus und schreibt das Ergebnis nach `z`. Danach setzt sie ihren Status auf `S_STOPPED` und signalisiert den auf sie wartenden Rechenauftrag *r*. Der Besitz des Mutex geht jetzt auf den wartenden Auftrag über. Dieser liest das Ergebnisregister, setzt den Status der Recheneinheit auf `S_READY` und gibt den Mutex *r.lock* danach explizit durch ein `pthread_mutex_unlock(`*r.lock*`)` wieder ab. Die Recheneinheit bekommt nun durch die Abgabe des Mutex wieder Rechenzeit. Sie nutzt diese jedoch erst dann, wenn sie den Zustand `S_FILLED` erreicht hat. Vorher stehen keine Aufträge zur Bearbeitung an. Deshalb legt sie sich auf *r.signal* schlafen, bis dieser Zustand erreicht ist.

Nachdem nun der Statuswechsel ausführlich erklärt wurde, folgen noch einige Erläuterungen zur Arbeitsweise eines Rechenauftrages. Ein Rechenauftrag muß, wie bereits erläutert, seinen arithmetischen Ausdruck auswerten. Dazu bekommt er den Parsebaum des Ausdrucks, d.h. einen Evaluationsplan, der von den Blättern zur Wurzel in Form eines geschachtelten Funktionsaufrufs der Operatorfunktionen bearbeitet wird. Für den Ausdruck $e = a + b * c$ wäre das zum Beispiel $plus(a, mul(b, c))$ in Präfixnotation.

Der Funktionsaufruf $plus(a, mul(b, c))$ wird nun von innen nach außen und von links nach rechts ausgewertet. Dabei werden alle Funktionen des Aufrufs sequentiell abgearbeitet. Für die Ausführung jeder einzelnen Funktion muß der Job folgende Schritte tun. Zuerst sucht er sich eine Recheneinheit, die die von ihm gewünschte Operation $o \in \{+, *\}$ ausführt. Die Information, welche Operation eine Recheneinheit ausführt, kann von allen Rechenaufträgen gelesen werden, da sie nicht mehr überschrieben wird. Ist eine Recheneinheit $r \in R_o$ $(R_+ = A, R_* = M)$ gefunden, so muß sie im Zustand `S_READY` sein. Um diese Information zu lesen, muß sich der Rechenauftrag jedoch den Mutex *r.lock* sichern, da ansonsten zwei Aufträge auf `S_READY` reagieren würden. Ist die Einheit nicht in diesem Zustand, wird der Mutex wieder abgegeben. Alle weiteren Schritte der Berechnung wurden bereits oben erklärt. Die Schritte, die ein Rechenauftrag machen muß, um eine Einheit für die von ihm gewünschte Operation zu bekommen, werden von der Funktion `eval()` ausgeführt. Der Aufruf `eval(ADD,x,y)` zum Beispiel sichert eine Recheneinheit $r \in R_+$ und liest das Ergebnis der Addition aus dem Register `z` der Einheit. Die Funktion `eval()` kann demnach die Operationen *plus* und *mul* nachbilden.

Hier nun die Implementierung des Modells:

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

```

[1] Bei einem pthread_cond_wait wird implizit der Mutex abgegeben, bis der Aufrufende durch ein pthread_cond_signal „geweckt" wird

```c
#define UNITS 2
#define JOBS 10
#define ADD 0
#define MUL 1

#define OK 0

#define S_READY   0
#define S_FILLED  1
#define S_STOPPED 2

/* Kommunikation laeuft ueber Registertupel */
struct u_tupel {
  int nr;
  pthread_mutex_t lock;
  pthread_cond_t  signal;
  int x; /* Input 1  */
  int y; /* Input 2  */
  int z; /* Ergebnis */
  int status; /* bereit, regsgefuellt, rechnend, fertig  */
  int type;   /* Unittyp: MUL ADD kann von jedem gelesen werden */
} unit_tupel[UNITS];

void
unit(struct u_tupel *my_tupel) {
  printf("This is Unit: %d\n",my_tupel->nr);
  while(1) {
    /* Setzen des Zustandsflags */
    pthread_mutex_lock(&my_tupel->lock);
    while(my_tupel->status!=S_FILLED)
      pthread_cond_wait(&my_tupel->signal,&my_tupel->lock);

    /* Rechnen */
    my_tupel->z=(my_tupel->type==ADD)?
      my_tupel->x+my_tupel->y: /* entweder addieren */
      my_tupel->x*my_tupel->y; /* oder multiplizieren */

    printf("Unit %d %s(%d,%d) \n",
    my_tupel->nr,
    (my_tupel->type==ADD)?"ADD":"MUL",
    my_tupel->x,my_tupel->y);

    /* Setzen des Zustandes */
    my_tupel->status=S_STOPPED;

    /* Signalisieren wartender Threads */
```

```
    pthread_mutex_unlock(&my_tupel->lock);
    pthread_cond_signal(&my_tupel->signal);
  }
}


int
eval(int op_kind,int x, int y){
  int nr,result;

  /* Suchen einer freien Recheneinheit */
  while(1) {
    nr=(++nr)%UNITS;
    if (unit_tupel[nr].type==op_kind) {
      pthread_mutex_lock(&unit_tupel[nr].lock);
      if (unit_tupel[nr].status!=S_READY) {
 pthread_mutex_unlock(&unit_tupel[nr].lock);
    sched_yield();
 }
      else break;
    }
  }

  /* Füllen der Register */
  unit_tupel[nr].x=x;
  unit_tupel[nr].y=y;
  unit_tupel[nr].status=S_FILLED;

  /* Anstoßen der Recheneinheit und Warten */
  pthread_cond_signal(&unit_tupel[nr].signal);
  while (unit_tupel[nr].status!=S_STOPPED)
    pthread_cond_wait(&unit_tupel[nr].signal,&unit_tupel[nr].lock);

  /* Lesen des Ergebnisses */
  result=unit_tupel[nr].z;
  unit_tupel[nr].status=S_READY;

  /* Freigeben der Recheneinheit */
  pthread_mutex_unlock(&unit_tupel[nr].lock);
  pthread_cond_signal(&unit_tupel[nr].signal);
  return result;
}

void
Job(int *nr) {
  int result;
```

```
  printf ("This is Job %d\n",*nr);
  switch((*nr)%3) {
    /* (-nr*3)+((nr*2)+5)*/
  case 0: result=eval(ADD,eval(MUL,-(*nr),3),
       eval(ADD,eval(MUL,*nr,2),5)); break;
    /* (8*2+5*-3)*nr */
  case 1: result=eval(MUL,eval(ADD,eval(MUL,8,2),eval(MUL,5,-3)),
      *nr); break;
    /* nr*3+1 */
  case 2: result=eval(ADD,eval(MUL,*nr,3),1); break;
  }
  printf("Job-Nr: %d Rechnung: %d Ergebnis: %d\n",*nr,(*nr)%3,result);
}

void
main() {
  pthread_t Th[JOBS];         /* Jobs            */
  pthread_t Units[UNITS];   /* Recheneinheiten */
  int JobNr[JOBS],nr;         /* Argumentliste   */

  /* Feld mit Threadnummern initialisieren */
  for(nr=0;nr<JOBS;JobNr[nr]=nr++);     /* JobNr-Liste */

  /* Starten der Recheneinheiten */
  for (nr=0;nr<UNITS;nr++){             /* Units initiieren */
    unit_tupel[nr].type=(nr%2)?ADD:MUL; /* Typ der Unit */
    unit_tupel[nr].nr=nr;               /* Unit-Nummer */
    unit_tupel[nr].status=S_READY;      /* Initialzustand ist "bereit" */
    pthread_mutex_init(&unit_tupel[nr].lock,NULL);  /* Mutex auf Unit */
    pthread_cond_init(&unit_tupel[nr].signal,NULL); /* Bedingungsvariable */

    pthread_create(&Units[nr], NULL, (void* (*)(void*))unit,  /* Start */
    (struct unit_t*) &unit_tupel[nr]);
 pthread_detach(Units[nr]);
  }

  /* Starten der Jobs */
  for (nr=0;nr<JOBS;nr++) { /* Jobs iniitiieren */
    pthread_create(&Th[nr], NULL, (void* (*) (void*))Job, (int *)&JobNr[nr]);
  }

 /* Warten auf Beendigung der Rechenjobs */
 for (nr=0;nr<JOBS;nr++){
    pthread_join(Th[nr],0);
 }
}
```

In `main()` werden zunächst einige Initialisierungen vorgenommen. Danach werden alle Threads gestartet. In den Zeilen 138 bis 140 erfolgt dann ein kontrolliertes Beenden der Applikation. Dazu wartet der `main()`-Thread mittels `pthread_join` auf die Beendigung der Rechenaufträge. Die Threads der Recheneinheiten wurden bewußt vom `main`-Thread entkoppelt (`pthread_detach`), da das Programm sonst aufgrund der Endlos-Iteration der Recheneinheiten (Warten auf Rechenaufträge) nicht beendet würde.

C.2. Beispielsitzung

Im folgenden soll demonstriert werden, wie das Debugging von POSIX-Threads mit Hilfe des modifizierten Gdb möglich ist. Dazu wird zunächst der Gdb mit dem in Abschnitt C.1 vorgestellten Programm *calc* gestartet.

```
% gdb calc
```

info-Befehle

Um Informationen über alle Threads zu erhalten, ist es nützlich, einen Breakpoint in Zeile 135 zu setzen. Wenn der Code dieser Zeile ausgeführt wird, sind bereits alle Threads erzeugt worden.
Jetzt wird das Programm gestartet, und bei Erreichen der Zeile 135 übergibt der Gdb dem Nutzer die Kontrolle. Mit dem Befehl **info pthreads** erhält man nun Informationen zu allen gestarteten Threads. Es werden Angaben über die Thread-ID `Id`, die Prozeß-ID des Threads `Pid`, die Threadadresse `Addr`, die Threadpriorität `Prio`, die dem Thread zugeordnete Funktion `entry` und ihr Argument `earg`, über einen eventuell blockierenden Mutex `MBO` oder eine Bedingungsvariable `CVWF` und den im TDI definierten und den in der Implementierung kodierten Zustand gemacht. Hier die Ausgabe:

(gdb) **info pthreads**

```
(POSIX Threads) thread information:

 Id  Pid   Addr      Prio     entry       entry_arg   MBO CVWF  State  RState
  12 15639 0xaf968      0             Job (0x1391c   )   -    - READY   [0x1208]
  11 15639 0xa40d8      0             Job (0x1391c   )   -    - READY   [0x1208]
  10 15639 0x98848      0             Job (0x1391c   )   -    - READY   [0x1208]
   9 15639 0x8cfb8      0             Job (0x1391c   )   -    - READY   [0x1208]
   8 15639 0x81728      0             Job (0x1391c   )   -    - READY   [0x1208]
   7 15639 0x75e98      0             Job (0x1391c   )   -    - READY   [0x1208]
   6 15639 0x6a608      0             Job (0x1391c   )   -    - READY   [0x1208]
   5 15639 0x5ed78      0             Job (0x1391c   )   -    - READY   [0x1208]
   4 15639 0x534e8      0             Job (0x1391c   )   -    - READY   [0x1208]
   3 15639 0x47c58      0             Job (0x1391c   )   -    - READY   [0x1208]
```

```
   2 15639 0x3c3c8      0           unit (0x13424   )   -   - READY   [0x120c]
   1 15639 0x30b38      0           unit (0x13424   )   -   - READY   [0x120c]
  13 15639 0x2ff70      0           main (0x0       )   -   - RUNNING [0x1209]
(gdb)
```

Informationen über die derzeit instantiierten Mutexe und Bedingungungsvariablen erhält man mit **info pmutex** und **info pcond**.

(gdb) **info pmutex**

```
(POSIX Threads) mutex information:

 ID    Addr   OwnerID
  1 0x2f4dc       -
  2 0x2f528       -
(gdb)
```

(gdb) **info pcond**

```
(POSIX Threads) condition variables information:

 ID    Addr   MutexID
  1 0x2f4fc       -
  2 0x2f548       -
(gdb)
```

Es gibt keinen Thread, der einen Mutex besitzt. Da bisher noch kein `pthread_cond_wait` ausgeführt wurde, gibt es keine Zuordunungen zwischen Mutexen und Bedingungsvariablen.

Threadspezifische Breakpoints

Da es sehr nützlich sein kann, einzelne Threads zu beobachten, können threadspezifische Breakpoints gesetzt werden. Will man wissen, wann die Recheneinheit, die von Thread 2 ausgeführt wird, ein Ergebnis berechnet, dann setzt man mit dem Befehl **break 37 pthread 2** einen Breakpoint für Thread 2 in Zeile 37. Anschließend führt man ein **continue** aus und prüft bei Erreichen des mit breakth erzeugten Breakpoints den Inhalt der dieser Recheneinheit zugeordneten Struktur.

(gdb) **break 37 pthread 2**

```
Breakpoint 2 at 0x134a4: file calc.c, line 37.
```

(gdb) **continue**

```
Continuing.
This is Unit: 0
This is Unit: 1
This is Job 0
This is Job 1
This is Job 2
This is Job 3
This is Job 4
This is Job 5
This is Job 6
This is Job 7
This is Job 8
This is Job 9
Unit 0 MUL(0,3)
Unit 0 MUL(0,2)
Unit 0 MUL(8,2)

Breakpoint 2, unit (my_tupel=0x2f524) at calcn.c:37
37          my_tupel->z=(my_tupel->type==ADD)?
(gdb)
```

(gdb) **p *my_tupel**

```
$1 = {nr = 1, lock = {queue = {head = 0x0, tail = 0x0}, lock = 1 '\001',
    owner = 0x3c3c8, flags = 1, prioceiling = 0, protocol = 0,
    prev_max_ceiling_prio = 0}, signal = {queue = {head = 0x47c58,
      tail = 0x47c58}, flags = 1, waiters = 1, mutex = 0x2f528}, x = 0, y = 5,
  z = 0, status = 1, type = 0}
(gdb)
```

Auch bei der Information über die derzeit gesetzten Breakpoints kann man Informationen darüber erhalten, ob ein Breakpoint threadspezifisch und bei welchem Thread gesetzt ist. Das zeigt die Ausgabe des Kommandos **info breakpoints**. Neu ist hierbei die Angabe, ob der Thread für einen POSIX-Thread gesetzt wurde.

(gdb) **info breakpoints**

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00013d40 in main at calcn.c:137
        breakpoint already hit 1 time
2   breakpoint     keep y   0x000134a4 in unit at calcn.c:37
        stop only in POSIX thread 2
        breakpoint already hit 1 time
```

Beobachtung des Schedulings mit „continue -cs“

Information darüber, welcher Thread als nächstes ausgeführt wird, kann man mit Hilfe des Befehls **continue -cs** erhalten. Das Programm wird hierbei wie bei einem **continue** fortgeführt, jedoch wird nach dem nächsten Kontextwechsel gestoppt, und die Kontrolle geht wieder an den Nutzer über.

(gdb) **continue -cs**

```
Continuing.
Unit 1 ADD(0,5)
context switch [OldID 2] -> [NewID 5]
(gdb)
```

Ein anschließendes **info pthreads** gibt uns aufschlußreiche Informationen über den Programmzustand.

(gdb) **info pthreads**

```
(POSIX Threads) thread information:

 Id  Pid   Addr      Prio     entry       entry_arg   MBO CVWF  State  RState
  12 15639 0xaf968      0            Job (0x1391c   )   -   - READY   [0x1208]
  11 15639 0xa40d8      0            Job (0x1391c   )   -   - READY   [0x1208]
  10 15639 0x98848      0            Job (0x1391c   )   -   - READY   [0x1208]
   9 15639 0x8cfb8      0            Job (0x1391c   )   -   - READY   [0x1208]
   8 15639 0x81728      0            Job (0x1391c   )   -   - READY   [0x1208]
   7 15639 0x75e98      0            Job (0x1391c   )   -   - READY   [0x1208]
   6 15639 0x6a608      0            Job (0x1391c   )   -   - READY   [0x1208]
   5 15639 0x5ed78      0            Job (0x1391c   )   -   - RUNNING [0x1208]
   4 15639 0x534e8      0            Job (0x1391c   )   -   - READY   [0x1208]
   3 15639 0x47c58      0            Job (0x1391c   )   -   - READY   [0x1208]
   2 15639 0x3c3c8      0           unit (0x13424   )   -   2 BLOCK C [0x1614]
   1 15639 0x30b38      0           unit (0x13424   )   -   1 BLOCK C [0x1614]
  13 15639 0x2ff70      0           main (0x0       )   -   - BLOCK O [0x1611]
(gdb)
```

Es ist zu erkennen, daß Thread 5 derzeit aktiv ist. Ein **info pcond** zeigt, daß aufgrund der blockierenden Rufe von `pthread_cond_wait` durch die Recheneinheiten eine Assoziation von Mutexen und Bedingungsvariablen erfolgt ist.

(gdb) **info pcond**

```
(POSIX Threads) condition variables information:

 ID    Addr   MutexID
  1 0x2f4fc       1
  2 0x2f548       2
(gdb)
```

Kontrollflußbindung mit ptattach

Zur Demonstration der Kontrollflußbindung werden alle Breakpoints gelöscht, und es wird ein Breakpoint bei der Funktion `Job` gesetzt. Danach wird das Programm neu gestartet.

(gdb) **delete 1 2**

(gdb) **break Job**

```
Breakpoint 3 at 0x13924: file calcn.c, line 96.
```

(gdb) **run**

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
TDI reinit
Starting program: /vol/eiche-vol3/dschulz/dipl_thesis/impl/TestTDI/calc
no reissue

POSIX Threads Library is TDI aware.
[Kernel Threads : User Level Threads] =  [1:N] i.e. FSU, MIT
Signal used for scheduling: 14
This is Unit: 0
This is Unit: 1

Breakpoint 3, Job (nr=0xeffffad0) at calc.c:96
96        printf ("This is Job %d\n",*nr);
(gdb)
```

Es ist hinzuzufügen, daß der Gdb aufgrund des Lesens der Variablen `pthread_debug_with_TDI` und `pthread_debug_TDI_sig_ignore` den Typ und das für das Scheduling genutzte Signal der POSIX-Threads-Implementierung feststellen kann. Beide Variablen werden im Laufe des Debuggings vom Gdb geschrieben. Ein **info pthreads** zeigt uns, daß der derzeitige Thread die ID 3 hat.

(gdb) **info pthreads**

```
(POSIX Threads) thread information:

 Id  Pid   Addr      Prio     entry       entry_arg   MBO CVWF  State  RState
  12 15725 0xaf968      0             Job (0x1391c   )   -    - READY   [0x1208]
  11 15725 0xa40d8      0             Job (0x1391c   )   -    - READY   [0x1208]
  10 15725 0x98848      0             Job (0x1391c   )   -    - READY   [0x1208]
   9 15725 0x8cfb8      0             Job (0x1391c   )   -    - READY   [0x1208]
   8 15725 0x81728      0             Job (0x1391c   )   -    - READY   [0x1208]
   7 15725 0x75e98      0             Job (0x1391c   )   -    - READY   [0x1208]
   6 15725 0x6a608      0             Job (0x1391c   )   -    - READY   [0x1208]
   5 15725 0x5ed78      0             Job (0x1391c   )   -    - READY   [0x1208]
   4 15725 0x534e8      0             Job (0x1391c   )   -    - READY   [0x1208]
   3 15725 0x47c58      0             Job (0x1391c   )   -    - RUNNING [0x1208]
   2 15725 0x3c3c8      0            unit (0x13424   )   -    2 BLOCK C [0x1614]
   1 15725 0x30b38      0            unit (0x13424   )   -    1 BLOCK C [0x1614]
  13 15725 0x2ff70      0            main (0x0       )   -    - BLOCK O [0x1611]
(gdb)
```

Wir selektieren den aktuellen Thread mittels **ptattach 3**.

(gdb) **ptattach 3**

Nun führen wir die Funktion `Job` schrittweise aus. Da der Thread 3 selektiert ist, wird die Applikation nach einem **next** immer wieder im Thread 3 stoppen. Wäre kein Thread selektiert, hätte **next** die gewohnte prozeßspezifische Semantik.

(gdb) **next**

```
This is Job 0
97        switch((*nr)%3) {
(gdb)
```

(gdb) **next**

```
99        case 0: result=eval(ADD,eval(MUL,-(*nr),3),
(gdb)
```

(gdb) **next**

```
This is Job 1
This is Job 2
This is Job 3
This is Job 4
This is Job 5
This is Job 6
This is Job 7
This is Job 8
This is Job 9
Unit 0 MUL(0,3)
Unit 0 MUL(0,2)
Unit 0 MUL(8,2)
Unit 1 ADD(0,5)
Unit 0 MUL(5,-3)
Unit 1 ADD(0,5)
100                         eval(ADD,eval(MUL,*nr,2),5)); break;
(gdb)
```

(gdb) **next**

```
107       printf("Job-Nr: %d Rechnung: %d Ergebnis: %d\n",*nr,(*nr)%3,result);
(gdb)
```

Es ist deutlich zu erkennen, daß während der Ausführung der geschachtelten `eval`-Funktion die Jobs 1 bis 9 aktiv waren. Ihre Kontrollflüsse wurden jedoch nicht betrachtet. Bei einem unmodifizierten Debugger hätte jeder dieser Threads in der Funktion `Job` gestoppt.

Es ist hinzuzufügen, daß die Kommandos **step** und **break** während der Thread-Selektion an den spezifizierten Thread gebunden sind. Die während der Threadselektion erzeugten Breakpoints werden automatisch an diesen Thread gebunden.

Eingreifen in das Scheduling

Die Befehle **ptsuspend** und **ptresume** erlauben es, die Ausführung eines Threads zu stoppen und fortzuführen. Bisher sind diese nur im Zusammenhang mit der FSU-Pthreads-Bibliothek benutzbar, da diese das Schreiben des Attributes `state` erlaubt.

Um die beiden Befehle zu demonstrieren, wird nach dem Start der Applikation (Breakpoint bei Job) der Thread 8 suspendiert. Danach wird der Kontrollfluß des Thread 10 selektiert. Anhand der Ausgabe des Programmes ist zu sehen, das Thread 8 noch nicht ausgeführt wurde. Thread 8 wird nun wieder aktiviert, und das Programm wird fortgeführt bis zur vollständigen Ausgabe der Job-Nummer in Thread 8. Dazu wird ein Breakpoint in Zeile 97 für Thread 8 gesetzt.

(gdb) **break Job**

```
Breakpoint 1 at 0x13924: file calcn.c, line 96.
(gdb)
```

(gdb) **run**

```
Starting program: /vol/eiche-vol3/dschulz/dipl_thesis/impl/TestTDI/calc
no reissue

POSIX Threads Library is TDI aware.
[Kernel Threads : User Level Threads] =  [1:N] i.e. FSU, MIT
Signal used for scheduling: 14
This is Unit: 0
This is Unit: 1

Breakpoint 1, Job (nr=0xeffffad0) at calcn.c:96
96        printf ("This is Job %d\n",*nr);
(gdb)
```

(gdb) **ptsuspend 8**

(gdb) **info pthreads**

```
(POSIX Threads) thread information:

 Id  Pid   Addr      Prio     entry       entry_arg   MBO CVWF  State  RState
  12 15755 0xaf968      0           Job (0x1391c   )   -   - READY   [0x1208]
  11 15755 0xa40d8      0           Job (0x1391c   )   -   - READY   [0x1208]
  10 15755 0x98848      0           Job (0x1391c   )   -   - READY   [0x1208]
   9 15755 0x8cfb8      0           Job (0x1391c   )   -   - READY   [0x1208]
   8 15755 0x81728      0           Job (0x1391c   )   -   - BLOCK S [0x9200]
   7 15755 0x75e98      0           Job (0x1391c   )   -   - READY   [0x1208]
   6 15755 0x6a608      0           Job (0x1391c   )   -   - READY   [0x1208]
   5 15755 0x5ed78      0           Job (0x1391c   )   -   - READY   [0x1208]
```

```
   4 15755 0x534e8      0           Job (0x1391c   )   -   - READY   [0x1208]
   3 15755 0x47c58      0           Job (0x1391c   )   -   - RUNNING [0x1208]
   2 15755 0x3c3c8      0          unit (0x13424   )   -   2 BLOCK C [0x1614]
   1 15755 0x30b38      0          unit (0x13424   )   -   1 BLOCK C [0x1614]
  13 15755 0x2ff70      0          main (0x0       )   -   - BLOCK O [0x1611]
(gdb)
```

(gdb) **ptattach 10**

```
Continuing.
This is Job 0
This is Job 1
This is Job 2
This is Job 3
This is Job 4
This is Job 6

(TDI) Thread 10 attached:
0x14f44 in pthread_body ()
(gdb)
```

(gdb) **ptresume 8**

(gdb) **d 1**

(gdb) **break 97 pthread 8**

```
Breakpoint 2 at 0x1393c: file calcn.c, line 97.
(gdb)
```

(gdb) **continue**

```
Continuing.
This is Job 7
This is Job 8
This is Job 9
Unit 0 MUL(0,3)
This is Job 5

Breakpoint 2, Job (nr=0xeffffae4) at calcn.c:97
97        switch((*nr)%3) {
(gdb)
```

Es ist zu beobachten, daß Thread 8 die Ausgabe „This is Job 5“ erst macht, nachdem alle anderen Jobs bereits ihre Ausgabe getätigt haben.

Anzeige der Ausführungsstacks

Die Ausführungsstacks verschiedener Threads lassen sich mit dem Befehl **ptstack** anzeigen. Zur Denmonstration wird wieder ein Breakpoint bei `Job` gesetzt und das Programm gestartet.

Ein **info pthreads** gibt folgende Ausgabe:

```
(gdb) info pthreads

(POSIX Threads) thread information:

 Id  Pid   Addr      Prio     entry       entry_arg   MBO CVWF  State  RState
  12 15773 0xaf968      0            Job (0x1391c   )   -    - READY   [0x1208]
  11 15773 0xa40d8      0            Job (0x1391c   )   -    - READY   [0x1208]
  10 15773 0x98848      0            Job (0x1391c   )   -    - READY   [0x1208]
   9 15773 0x8cfb8      0            Job (0x1391c   )   -    - READY   [0x1208]
   8 15773 0x81728      0            Job (0x1391c   )   -    - READY   [0x1208]
   7 15773 0x75e98      0            Job (0x1391c   )   -    - READY   [0x1208]
   6 15773 0x6a608      0            Job (0x1391c   )   -    - READY   [0x1208]
   5 15773 0x5ed78      0            Job (0x1391c   )   -    - READY   [0x1208]
   4 15773 0x534e8      0            Job (0x1391c   )   -    - READY   [0x1208]
   3 15773 0x47c58      0            Job (0x1391c   )   -    - RUNNING [0x1208]
   2 15773 0x3c3c8      0           unit (0x13424   )   -    2 BLOCK C [0x1614]
   1 15773 0x30b38      0           unit (0x13424   )   -    1 BLOCK C [0x1614]
  13 15773 0x2ff70      0           main (0x0       )   -    - BLOCK O [0x1611]
(gdb)
```

Wir wollen den Stack des Threads 2 (Recheneinheit) und des Threads 12 betrachten.

```
(gdb) ptstack 2

Thread 2: sp: 0x47ac0 pc: 0x15cc0
#0  0x15cc0 in pthread_cond_wait ()
#1  0x1349c in unit (my_tupel=0x2f524) at calcn.c:34
#2  0x14f5c in pthread_body ()
(gdb)
```

```
(gdb) ptstack 12

Thread 12: sp: 0xbb1b0 pc: 0x14f44
#0  0x14f44 in pthread_body ()
(gdb)
```

Man sieht, daß Thread 2 in einem Ruf von `pthread_cond_wait` blockiert ist und Thread 12 am Anfang seiner Ausführung steht