Studienarbeit:

POSIX Threads Debugger

von

Daniel Schulz

Humboldt Universität zu Berlin,
Institut für Informatik,
1997

Betreuer: Frank Müller, Ph.D.

1 Zusammenfassung

Gegenstand dieser Arbeit ist eine Erweiterug der Funktionalität des GNU-Debuggers Gdb, um das Debugging nebenläufiger Programme zu ermöglichen. Die Standardisierung einer Thread-Schnittstelle für die Programmiersprache C als POSIX-Threads ist ein wesentlicher Aspekt für die Popularisierung der Techniken des nebenläufigen Programmierens. Ein Problem besteht darin, daß die Fehlerbeseitigung mit einem herkömmlichen Debugger nur eingeschränkt möglich ist. Die Existenz von Threads ist für den Debugger nicht erkennbar und der Kontrollfluß eines einzelnen Threads kann nicht verfolgt werden. Änderungen am GNU-Debugger und eine Erweiterung der Standardschnittstelle der POSIX-Threads sind notwendig, um das gewohnte komfortable Arbeiten mit dem Gdb zu ermöglichen.
In der vorliegenden Arbeit werden Ansätze zur Lösung der Probleme erarbeitet und umgesetzt. Dazu ist eine gründliche Einarbeitung in den Aufbau und die Funktionsweise des Gdb sowie der POSIX-Threads am Beispiel der FSU-Pthreads-Bibliothek erforderlich, um Ansatzpunkte für Lösungen zu finden und deren Machbarkeit abzuwägen. Der Gdb und die Bibliothek erfuhren einige Änderungen, von denen ein Teil schon in der Praxis angewendet werden kann. Es ist nun möglich, Informationen über alle gestarteten Threads zu erhalten, threadspezifische Breakpoints zu setzen und Kontextwechsel zu erkennen und zu kontrollieren. Weiterhin wurden Befehle hinzugefügt, die noch experimentellen Charakter haben. Hierzu gehören solche, mit denen der Nutzer in das Scheduling der Threads eingreifen kann und ein Befehl, mit dessen Hilfe man die nächste Quelltextzeile erreicht, ohne den aktuellen Thread zu verlassen. Ergebnis dieser Arbeit ist ein Satz von Befehlen, der dem Gdb zugefügt wurde und ein Thread-Debugging ermöglicht. Die erarbeitete Lösung ist jedoch eingeschränkt auf die FSU-Pthreads-Bibliothek und die Betriebssysteme SunOS und Linux. Diese Einschränkungen sollen im Rahmen einer anschließenden Diplomarbeit untersucht und beseitigt werden.

Inhaltsverzeichnis

2 Einführung

Die nebenläufige und parallele Programmierung eröffnet neue Möglichkeiten, Algorithmen zu entwerfen und reale Abläufe zu modellieren. Ein mit dieser Programmiertechnik verbundenes Problem ist das Debugging nebenläufiger und paralleler Programme.
Threads sind eine Abstraktion auf Programmebene, durch die Nebenläufigkeit ausgedrückt wird. Diese Arbeit beschäftigt sich mit dem Vorgang der Fehlerbeseitigung in Programmen, in denen Threads verwendet werden. Auf die nebenläufige Programmierung soll nicht eingegangen werden.
Im Software-Entwicklungszyklus werden rund 50 Prozent der Gesamtzeit für das Debugging verwendet [1]. Daraus ergibt sich die Notwendigkeit, ein Werkzeug zu entwickeln, mit dem ein effektives Debugging möglich ist. Gegenstand dieser Arbeit ist die Modifikation einer bereits existierenden Debugger-Implementierung für sequentielle Programme und einer Thread-Bibliothek. Die Konzentration auf die Modifikation des Gdb bezüglich Thread-Bibliotheken ist berechtigt, da bisher nur wenige Betriebssysteme mit Kernel-Threads existieren. Eines davon ist Mach, und hierfür ist bereits eine Änderung des Gdb vorhanden [2]. Die letzten Versionen des Betriebssystems Linux sehen ebenfalls eine Unterstützung von Kernel-Threads vor. Hier gibt es jedoch noch Konflikte mit dem Gdb [3].
Um einen Debugger für die speziellen Anforderungen der nebenläufigen Programmierung zu implementieren, wurde ein bestehender Debugger erweitert, um möglichst schnell ein Ergebnis zu erzielen und die schon bestehende Nutzerschaft dieses Werkzeugs für diese Änderungen zu gewinnen. Die Wahl fiel dabei auf den Gdb - GNU Debugger. Als Thread-Implementierung stand die FSU-Pthreads-Bibliothek (im folgenden nur Pthreads genannt) der Florida State University zur Verfügung [6].
Diese Arbeit ist in folgende Abschnitte gegliedert. Der Abschnitt 3 soll die zentralen Begriffe dieser Arbeit und deren hier verwendete Semantik erläutern. In Abschnitt 4 wird es darum gehen, welche grundlegenden Eigenschaften Debugger und Bibliothek haben sollten, um an das Debugging nebenläufiger Einheiten angepaßt zu werden. In Abschnitt 5 werden Probleme dargestellt, die sich ergeben, wenn man Threads mit einem herkömmlichen Debugger analysieren und kontrollieren will. Anhand dieser Probleme werden Vorschläge und Ziele für Änderungen erarbeitet. Um ein zukünftiges Arbeiten mit dem geänderten Gdb zu illustrieren, wird in Abschnitt 5.3 dessen Anwendung auf ein Beispielprogramm gezeigt. Welche Änderungen und wie diese erfolgten,

behandelt der Abschnitt 6. Dabei wird näher auf die Semantik der neuen Befehle eingegangen, und Implementierungsansätze werden beschrieben. Die spezifischen Eigenschaften des Gdb und der Pthreads, die hierbei zu beachten sind, werden erläutert. Abschnitt 7 zeigt die in diesem Zusammenhang aufgetretenen architekturspezifischen Probleme. Im Abschnitt 8 werden bisher auf dem Gebiet des Debuggings nebenläufiger Programme erschienene Arbeiten vorgestellt und mit der hier vorliegenden Arbeit verglichen. Der Ausblick in Abschnitt 9 soll die zukünftig angestrebten Ziele darlegen.

3 Terminologie

In der Informatik ist die Verwendung von Begriffen aus dem Englischen sehr verbreitet. Da diese Studienarbeit jedoch an einer deutschen Universität verfaßt worden ist, mußte versucht werden, so viele Begriffe aus dem Deutschen zu verwenden, wie möglich.
Die Begriffe, zu denen es keine prägnante deutsche Entsprechung gibt, wurden englisch belassen. Dazu gehören Debugger, Debugging, Thread, Scheduler, Scheduling und Breakpoint. Probleme gab es bei Verben, deren Konjugation im Deutschen nicht festgeschrieben ist. Beispielhaft hierfür ist das Verb debuggen, dessen dritte Form im Singular nicht spezifiziert ist (er/sie/es debugt, debuggt, debugged, debugget).
Nun aber zur Definition einiger in dieser Arbeit verwendeter Begriffe.
Debugger:
Ein Debugger ist ein Programm, mit dessen Hilfe man die Fehler in einem anderen Programm feststellen kann. Der Debugger erlaubt dem Programmierer, den Programmzustand zu erfassen und zu ändern. Hierzu gehört, daß man alle Programmvariablen lesen und auch schreiben kann. Desweiteren kann man auch die Dynamik eines Programmes beobachten und gegebenenfalls ändern. Die Beobachtung wird durch die Information über den derzeitigen Programmzustand, d.h. die Gesamtheit der Werte der Variablen, ermöglicht. Man spricht von einem Quelltextdebugger (Source Code Debugger), wenn dem Nutzer zum derzeitigen Programmzähler (*program counter, pc*) die korrespondierende Zeilennummer des Quelltextes des Programms geliefert werden kann. Beeinflussen kann man die Dynamik durch ein Umsetzen des *pc* oder durch die Veränderung der Werte von Booleschen Ausdrücken, wie sie bei Schleifen oder if-then-else Konstrukten vorkommen.
Prozeß:
Prozesse stehen im folgenden immer für UNIX-Prozesse. Ein Prozeß ist dabei die spezifische Laufzeitumgebung eines Programmes. Ein Prozeß hat einen eigenen Kontrollfluß und eigenen Speicher. Er wird über eine Prozeßnummer identifiziert. Prozesse werden vom System geschedult, d.h sie erhalten CPU-Zeit nach einem vom System vorgegebenen Plan. Wenn ein Prozeß die CPU wieder abgeben muß, dann findet ein Kontextwechsel statt. Hierbei müssen alle prozeßrelevanten Informationen gespeichert werden.
Thread:
Threads sind nebenläufige Einheiten, die gemeinsamen Speicher, aber getrennte Kontrollflüsse haben. Bei der betrachteten Bibliotheksimplementie-

rung der POSIX-Threads sind Threads immer einem Prozeß zugeordnet, soweit es sich um ein UNIX-ähnliches System handelt. Threads sind im Gegensatz zu Prozessen „leichtgewichtiger", d.h die Thread-relevanten Informationen, die für die Ausführung bzw. Unterbrechung relevant sind, haben einen kleineren Umfang. Da Threads einem Prozeß zugeordnet sind, muß man bei einem Thread-Kontextwechsel keine prozeßspezifischen Daten speichern. Wenn der zugeordnete Prozeß die CPU abgeben muß, dann werden vom System alle prozeßspezifischen Daten gerettet. Threads können darüber hinaus Multiprozessoren mit gemeinsamen Speicher ausnutzen, um die Programmausführung zu beschleunigen. Dieser Aspekt wird jedoch nicht Gegenstand der vorliegenden Arbeit sein.

Scheduler:
Ein Scheduler ist ein Programm, das Betriebsmittel (z.B. CPU, Festplatte) mit begrenzter Zahl an gleichzeitigen Nutzern (z.B. ein Prozeß auf einer CPU zu einer Zeit) zuteilt.
Der Scheduler, der im folgenden betrachtet wird, teilt die CPU-Zeit den einzelnen Threads zu. Er arbeitet auf Warteschlangen, in die alle Threads eingetragen werden, die CPU-Zeit brauchen. Es können verschiedene Schedulingstrategien verfolgt werden (siehe dazu [6]).

Mutex:
Als Mutex wird hier das Objekt bezeichnet, das zur Realisierung eines gegenseitigen Ausschlusses („mutual exclusion") benutzt wird. Durch ein mutex_lock kann dieser Mutex (für kurze Abschnitte) von einem Thread angefordert werden und mittels mutex_unlock wieder abgegeben werden.

Bedingungsvariable („condition variable"):
Bedingungsvariablen dienen der Synchronisation zweier oder mehrerer Threads. Wenn eine Bedingung eintritt, die im allgemeinen von einem Thread hervorgerufen wird, so signalisiert dieser die auf diese Bedingung wartenden anderen Threads. Vor der Signalisierung warten die restlichen Threads auf einer sogenannten Bedingungsvariablen. Bedingte Synchronisation in dieser Form dient, im Gegensatz zum kurzzeitigen gegenseitigen Ausschluß, längerem Warten auf einen Zustandwechsel.

4 Voraussetzungen

Ziel dieser Arbeit ist es, Änderungen an einem existierenden Debugger vorzunehmen, die das Debugging nebenläufiger Einheiten ermöglichen. Der hier verwendete Debugger ist der Gdb. Um Änderungen auch an einem anderen Debugger durchzuführen, sollen hier dessen grundlegende Eigenschaften dargestellt werden, die für ein effizientes Debugging notwendig sind. Weiterhin sollen auch Anforderungen an eine Thread-Implementierung festgelegt werden. Hier werden die FSU-Pthreads betrachtet. Sollten die Änderungen des Gdb im Hinblick auf ein Thread-Debugging auch auf andere Implementierungen angewendet werden, so ist die Erfüllung dieser Mindestanforderungen der Schlüssel zu einer einheitlichen Schnittstelle für alle Thread-Implementierungen.

4.1 Anforderungen an einen Debugger

Bei dem zugrundegelegten Debugger handelt es sich um einen Breakpoint-Debugger. Er soll im einfachsten Fall die Applikation starten, weiterlaufen lassen, stoppen und sogenannte Breakpoints an beliebigen Stellen im Code setzen können. Um ein Arbeiten mit dem Quelltext zu ermöglichen, sollte er auch Informationen über Quellzeilennummern und deren Assemblercodeentsprechungen haben, um eine effektive Arbeit zu ermöglichen. Variablen, Funktionen und Typen müssen dem Debugger bekannt sein. Weiterhin wird vorausgesetzt, daß die Grundlage ein UNIX-ähnliches Betriebssystem mit Multiprozeß-Unterstützung ist. Hier können zur Umsetzung dieser minimalen Voraussetzungen die Systemrufe `fork()`, `execve()`, `ptrace()`, `kill()` etc. oder anderslautende in ihrer Funktionalität gleichartige benutzt werden. Der Debugger kann mittels `fork()` einen neuen Prozeß starten und diesem wird mittels `excve()` die zu debuggende Applikation zugeordnet (siehe Abbildung 1). Mit dem Systemruf `ptrace()` kann man im Kind-Prozeß alle Daten im Code- und Datenteil lesen und schreiben. Breakpoints setzt man vereinfacht folgendermaßen. Man kann eine spezielle Trap-Instruktion an jeder beliebigen Stelle im Codesegment einfügen, an der die Applikation, wenn sie im Laufe ihrer Ausführung einmal an diese Stelle kommt, stoppt.

Zusätzlich zu diesen minimalen Anforderungen an den Debugger soll es noch möglich sein, Funktionen der Applikation oder einer dazugehörigen Bibliothek zu nutzen. Dies ist durch die Verwendung von `ptrace()` möglich. Es kann Code für einen Funktionsaufruf in das Codesegment des Kind-Prozesses geschrieben werden, der dann von der Applikation gewissermaßen "ferngelenkt" ausgeführt wird. Man hat so die Möglichkeit, alle Funktionen einer mit der Anwendung verbundenen Bibliothek für Debugging-Zwecke zu nut-

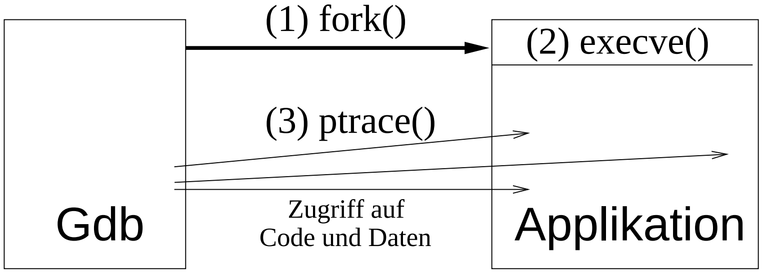

Abbildung 1: *Debugger und Kindprozeß*

zen. Insbesondere ist man in der Lage, auf die Rückgabewerte zuzugreifen. Implementiert man eine Thread-Umgebung, so kann man in ihr Funktionen bereitstellen, die der Nutzer zum Zwecke des Debugging aufrufen kann. Da dem Debugger diesbezüglich nur wenig Code hinzugefügt werden muß, wird seine Struktur kaum geändert.

4.2 Anforderungen an eine Thread-Implementierung

Eine Thread-Implementierung, die eine Reihe von Funktionen und Datenstrukturen zur Programmierung nebenläufiger Einheiten zur Verfügung stellt, sollte folgende grundlegende Ansatzpunkte für ein Debugging bieten.
Alle Threads, die während des Applikationslaufes erzeugt werden, sollten sichtbar sein, d.h. es sind Datenstrukuren oder Funktionen vorhanden, die Informationen über die bereits erzeugten Threads liefern. Desweiteren sind Schedulinginformationen für den Programmierer sehr wichtig. Daraus ergibt sich die Notwendigkeit, die Liste der wartenden Threads, deren Priorität und gewisse Statusinformationen zur Verfügung zu stellen. Als Beispiel für eine derartige Darstellung der geschedulten Einheiten ist das UNIX-Shell-Kommando `ps` anzugeben, es werden hier jedoch Prozeß- und keine Thread-Informationen bereitgestellt. Es kann als Maßstab für die Ausführlichkeit der Statusinformationen herangezogen werden.
Ebenfalls von Interesse sind Informationen über den derzeitigen Kontext eines Threads (letzter *pc*, sonstiger Registerstatus etc.) oder über Objekte, die der Synchronisation von Threads dienen (Semaphore etc.).
Zentraler Bestandteil einer Thread-Implementierung ist der Scheduler. Er sorgt dafür, daß gewisse Scheduling-Protokolle eingehalten werden. Es sollte in der Thread-Implementierung eine Schnittstelle geben, die eine Steuerung

des Schedulers von außen ermöglichen. Damit könnten einzelne Threads aus den Schedulinglisten entfernt werden oder wieder an beliebiger Stelle eingefügt werden. Desweiteren sollten Änderungen der Reihenfolge von Threads in beliebigen Warteschlangen möglich sein.

5 Problembeschreibung

Um Ansatzpunkte für eine Lösung des Problems Thread-Debugging zu finden, sollen hier die Unterschiede zwischen dem Debugging sequentieller und nebenläufiger Programme herausgearbeitet werden. Auf dieser Grundlage werden konkrete Zielsetzungen in Form neuer Befehle für den Gdb abgeleitet.

5.1 Thread-Debugging

Bevor man sich die Frage stellt, um welche neuen Kommandos man einen Debugger erweitert, sollte man grundsätzliche Eigenschaften dieser Kommandos festlegen. Es sollten Basisbefehle sein, die genügend Funktionalität bieten, um einzeln Anwendung zu finden, aber auch in Kombination mit anderen Kommandos eine erweiterte Funktionalität bieten. Alle neuen Befehle sollten aber auch in das Schema der bereits vorhandenen Befehle passen. Es sollten keine Namensähnlichkeiten neuer Kommandos mit alten bestehen, um den Benutzer nicht zu irritieren. Neue Befehle, die Spezialfälle bereits vorhandener Kommandos sind, sollten trotzdem für den Allgemeinfall die gleiche Semantik haben.
Nun aber zur eigentlichen Fragestellung, welche Kommandos man hinzufügen sollte. Hierzu ist es nötig, die Unterschiede zwischen sequentiellen und nebenläufigen Programmen zu beleuchten.
Ein sequentielles Programm hat nur einen Kontrollfluß, so daß alle jemals ausgeführten Teile des Programms nacheinander ausgeführt werden. Bei einem nebenläufigen Programm gibt es mehrere quasi parallel existierende Kontrollflüsse. Jeder Thread repräsentiert einen dieser Kontrollflüsse. Da bei den meisten Rechnern weniger CPU's als Threads vorhanden sind, teilen sich mehrere Threads eine CPU. Jeder Thread bekommt einen Teil der CPU-Zeit. Die Aufteilung der CPU-Zeit kann in verschiedener Weise erfolgen. Zum einen kann man jeden Thread seine Instruktionen ausführen lassen, bis man den nächsten startet (sequentielle Ausführung); die andere Variante ist, jeden Thread einen Teil seiner Instruktionen ausführen zu lassen und ihn dann

wieder auf CPU-Zeit warten zu lassen. Die Schwierigkeiten, die sich beim Debuggen nebenläufiger Programme ergeben, werden hier bereits ersichtlich, und mit einem herkömmlichen Debugger für sequentielle Programme ergeben sich gewisse Probleme.

Zum einen gibt es ein Problem, wenn zwei Threads dasselbe Stück Programmcode ausführen. Setzt man in diesem Programmstück einen Breakpoint, so ist nicht klar, welcher Thread dort zuerst stoppt. Eine Forderung besteht also darin, threadspezifische Breakpoints setzen zu können. So ist es dann möglich, einzelne Threads zu debuggen.

Ein weiteres Problem im Zusammenhang mit den angesprochenen parallelen Kontrollflüssen besteht beim `next`-Kommando. Seien also wieder zwei Threads gegeben, denen dasselbe Codestück zugeordnet ist. Befindet man sich nun an einem Breakpoint und führt ein herkömmliches `next` aus, so stoppt der Debugger das Programm zwar in der nächsten ausgeführten Zeile, zwischenzeitlich kann jedoch wieder ein Kontextwechsel erfolgt sein und man debugt jetzt einen ganz anderen Thread. Es wird deswegen ein Kommando benötigt, mit dem ein `next` ausgeführt wird, welches wieder im selben Thread stoppt.

Weiterhin kann es von Vorteil sein, an einem Kontextwechsel zu stoppen. Dazu kann eine Variante des continue-Befehls dienen.

Der Nutzer hat eventuell auch Interesse, aktiv in das Scheduling einzugreifen. Dazu könnte gehören, einen wartenden Thread zu aktivieren sowie Threads ganz aus den Scheduling-Warteschlangen zu entfernen und wieder an anderer Stelle einzufügen.

5.2 Ziele

Ergebnis dieser Arbeit soll es sein, eine Sammlung von Debugging-Kommandos für den Gdb zu implementieren, die den im letzten Abschnitt beschriebenen Anforderungen entsprechen. Hierfür seien folgende Kommandos vorgeschlagen (in Gdb-Kommandosyntax):

- **info threads**:
  Status aller Threads;

- **continue -cs**:
  continue mit Stopp bei Kontextwechsel;

- **nexttth**:
  next mit Ziel im aktuellen Thread;

- **breakth** [*tid*]+ [*Location*]:
  threadspezifischer Breakpoint bei *Location*;

- **switchth** *tid*:
  unbedingter Kontextwechsel zu Thread *tid*;

- **suspendth** [*tid*]+ [*Q*]:
  Threads aus Queue Q entfernen;

- **resumeth** [*tid*]+ [*Q*] [*Position*]:
  Threads wieder in Queue Q an der Stelle *Position* einfügen.

5.3 Ein Anwendungsbeispiel

Um die Änderungen an einem Thread-Debugger klar herauszuarbeiten, ist es vorteilhaft, eine exemplarische Applikation zu betrachten, die die für das nebenläufige Programmieren wesentlichen Funktionen der POSIX-Threads benutzt.

5.3.1 Aufbau und Funktionsweise des Beispielprogramms

Es soll ein System modelliert werden, in dem es beschränkte Betriebsmittel (Recheneinheiten) gibt und demgegenüber eine größere Menge an Nutzern (Rechenaufträge) dieser Betriebsmittel vorhanden ist. Gegeben sei eine Menge von Recheneinheiten R. Jede dieser Recheneinheiten kann genau eine der folgenden Operationen ausführen:

- Multiplikation zweier ganzer Zahlen

- 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.
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 eingetroffen 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, 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, 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

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

$r \in R_o$ $(R_+ = A, R_* = M)$ 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>

% weitere #includes und #defines %

pthread_mutex_t CountLock;
pthread_cond_t  CountSignal;
int JobCount=0;

struct u_tupel {

 % Beschreibung im Text  %

} unit_tupel[UNITS];

void
unit(struct u_tupel *my_tupel) {
  printf("This is Unit: %d\n",my_tupel->nr);
  while(1) {
    pthread_mutex_lock(&my_tupel->lock);
    while(my_tupel->status!=S_FILLED)
      pthread_cond_wait(&my_tupel->signal,&my_tupel->lock);
    my_tupel->z=(my_tupel->type==ADD)?
      my_tupel->x+my_tupel->y: /* entweder addieren */
      my_tupel->x*my_tupel->y; /* oder multiplizieren */
    my_tupel->status=S_STOPPED;
    pthread_mutex_unlock(&my_tupel->lock);
    pthread_cond_signal(&my_tupel->signal);
  }
}

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

```
  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);
        pthread_yield(NULL);
      }
      else break;
    }
  }

  unit_tupel[nr].x=x;
  unit_tupel[nr].y=y;
  unit_tupel[nr].status=S_FILLED;
  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);
  result=unit_tupel[nr].z;
  unit_tupel[nr].status=S_READY;
  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);
  pthread_mutex_lock(&CountLock);
  JobCount++;
  pthread_mutex_unlock(&CountLock);
  pthread_cond_signal(&CountSignal);
```

```
92  }
93
94  void
95  main() {
96    pthread_t Th; /* Hier wird Adresse der Thread-Struktur abgelegt, diese
97    * wird aber im Programm nicht verwendet, deshalb kann
98    * sie auch ueberschrieben werden bei jedem pthread_create */
99    int kind_of[]={ADD,MUL},nr;
100   int JobNr[JOBS];
101   pthread_init();
102   for(nr=0;nr<JOBS;JobNr[nr]=nr++);     /* JobNr-Liste */
...
105   for (nr=0;nr<UNITS;nr++){
.
.      % Initialisierung der Unit (Unittyp, Startzustand etc.)
.
111     pthread_create(&Th,NULL,(pthread_func_t) unit,    /* Unit starten */
112     (struct unit_t*) &unit_tupel[nr]);
113   }
114   for (nr=0;nr<JOBS;nr++) { /* Jobs iniitiieren */
115     pthread_create(&Th,NULL,(pthread_func_t) Job, (int *)&JobNr[nr]);
116   }
117   pthread_mutex_lock(&CountLock);
118   while(JobCount!=JOBS)           /* wenn nicht alle fertig, schlafe ... */
119     pthread_cond_wait(&CountSignal,&CountLock);
120   exit(0);                        /* ... sonst exit */
121 }
```

In `main()` werden zunächst einige Initialisierungen vorgenommen. Danach werden alle Threads gestartet. In den Zeilen 118 bis 121 erfolgt dann ein kontrolliertes Beenden der Applikation. Dazu wartet der `main()`-Thread, bis ihn alle Rechenaufträge über die Bedingungsvariable `CountSignal` signalisiert haben. Jetzt kann das Programm mit `exit()` beendet werden.

5.3.2 Anwendung des Gdb als Thread-Debugger

Nun soll gezeigt werden, wie das Debuggen von Threads mit Hilfe des Gdb möglich ist. Dazu wird zunächst der Gdb mit dem in Abschnitt 5.3.1 vorgestellten Programm *calc* gestartet.

```
% gdb calc
```

Um Informationen über alle Threads zu erhalten, ist es nützlich, einen Breakpoint in Zeile 117 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 117 übergibt der Gdb dem Nutzer die Kontrolle. Mit dem Befehl **info threads** erhält man nun Informationen zu allen gestarteten Threads. Es werden Angaben über Threadnummer `TID`, Threadadresse `ADDR`, Status `S` (*R=ready, B=blockiert*), Threadprioritäten `Base, Max, New`, Mutex `M`, Bedingungsvariablen `C` und über die dem Thread zugeordnete Funktion `func` und die Adresse des Argumentes `arg` gemacht. Hier die Ausgabe:

```
Breakpoint 1, main () at fifo_calc.c:117
117       pthread_mutex_lock(&CountLock);
```

(gdb) **info threads**

```
Current thread: 0

+-- ID's --+S+-  Priority  -+-------- Misc -----------+
+TID ADDR    +Base  Max  New+func arg
 15  8099800 R  C:-  M:- 0   -1  -1  Job 0xbffff654
 14  8099000 R  C:-  M:- 0   -1  -1  Job 0xbffff650
 13  8090800 R  C:-  M:- 0   -1  -1  Job 0xbffff64c
 12  8090000 R  C:-  M:- 0   -1  -1  Job 0xbffff648
 11  8087800 R  C:-  M:- 0   -1  -1  Job 0xbffff644
 10  8087000 R  C:-  M:- 0   -1  -1  Job 0xbffff640
 9   807e800 R  C:-  M:- 0   -1  -1  Job 0xbffff63c
 8   807e000 R  C:-  M:- 0   -1  -1  Job 0xbffff638
 7   8075800 R  C:-  M:- 0   -1  -1  Job 0xbffff634
 6   8075000 B  C:x  M:x 0   -1  -1  Job 0xbffff630
 5   806c800 B  C:x  M:x 0   -1  -1  unit 0x805802c
 4   806c000 R  C:x  M:x 0   -1  -1  unit 0x8057fe0
 3   8063800 B  C:x  M:x 0   -1  -1  unit 0x8057f94
 2   8063000 B  C:x  M:x 0   -1  -1  unit 0x8057f48
 1   805c800 B  C:x  M:x 101 -1  -1  pthread_p_select 0x40005c40
 0   805c000 R B  C:-  M:- 0   -1  -1  main 0x0
(gdb)
```

Da es sehr nützlich sein kann, einzelne Threads zu beobachten, können threadspezifische Breakpoints gesetzt werden. Will man wissen, wann die Recheneinheit, die in Thread 5 ausgeführt wird, ein Ergebnis berechnet, dann setzt man mit dem Befehl **breakth 5 35** einen Breakpoint für Thread 5 in Zeile 35. Anschließend führt man ein **continue** aus und prüft bei Erreichen des mit breakth erzeugten Breakpoints den Inhalt der dieser Recheneinheit zugeordneten Struktur.

(gdb) **breakth 5 35**

```
Breakpoint 2 at 0x8048f6c: file fifo_calc.c, line 35.
```

(gdb) **continue**

```
Continuing.
Unit 2 MUL(0,2)
Unit 0 MUL(5,-3)
Unit 2 MUL(2,3)
Unit 1 ADD(0,5)
Unit 0 MUL(8,2)
Unit 2 MUL(0,3)

Breakpoint 3, unit (my_tupel=0x805802c) at fifo_calc.c:35
35          my_tupel->z=(my_tupel->type==ADD)?
```

(gdb) **p *my_tupel**

```
$2 = {nr = 3, lock = {queue = {head = 0x0, tail = 0x0}, lock = 1 '\001',
    owner = 0x806c800, flags = 1, prio_ceiling = 0, protocol = 0,
    prev_max_ceiling_prio = 0}, signal = {queue = {head = 0x807e000,
      tail = 0x807e000}, flags = 1, waiters = 1, mutex = 0x8058030}, x = 6,
  y = 1, 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 der Threadnummer `TID`.

(gdb) **info breakpoints**

```
Num TID Type           Disp Enb Address    What
1   all breakpoint     keep y   0x080495c0 in main at fifo_calc.c:117
        breakpoint already hit 1 time
2   5   breakpoint     keep y   0x08048f6c in unit at fifo_calc.c:35
        breakpoint already hit 1 time
```

Mit der Ausführung der nächsten Befehle soll demonstriert werden, welche Möglichkeiten durch die neu eingeführten Kommandos eröffnet werden. Die 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

Breakpoint 4 at 0x80543a8.
Breakpoint 5 at 0x804977c.
Continuing.
Unit 3 ADD(6,1)
context switch:T 5 (806c800)-->T 2 (8063000)
(gdb)
```

Die Befehle **switchth**, **suspendth**, **resumeth** und **nextth** können an dieser Stelle noch nicht demonstriert werden, da sie noch nicht vollständig implementiert sind.

6 Entwurf und Implementierung

Im Abschnitt 5.2 ist ein für das Thread-Debugging hilfreicher Satz an neuen Befehlen für den Gdb als Zielsetzung erarbeitet worden. In diesem Abschnitt wird die Implementierung jedes Befehls erörtert. Hierbei werden unter anderem die dabei umzusetzende Semantik des Befehls und auftretende Schwierigkeiten dargestellt. Vorher werden noch einige allgemeine Aspekte des Gdb und der Bibliothek im Zusammenhang mit der Implementierung dargelegt.

6.1 Änderungen am Debugger

Es wurde bereits erwähnt, welche Voraussetzungen ein Debugger erfüllen muß, um zusammen mit einer Bibliotheksimplementierung ein Thread-Debugging zu ermöglichen. Da der Gdb diese Anforderungen erfüllt und darüber hinaus in der UNIX-Welt einer der meistverwendeten Debugger ist, könnten Modifikationen in Bezug auf Threads einer großen Nutzerschaft dienen. Weiterhin sind alle Quellen dieses mächtigen Werkzeugs frei verfügbar. Der Gdb ist auf den verschiedensten Architekturen und Betriebssystemen lauffähig und unterstützt eine Reihe von Object-File-Formaten (COFF, ELF, a.out,

IEEE, Oasys).
Die Quelldateien (20Mb) sind so strukturiert, daß eine Modifikation nach einer gewissen Einarbeitungszeit möglich ist. Da die Zahl der Programmierer, die Modifikationen am Gdb vorgenommen haben, sehr hoch ist, muß man aber Abstriche bei der Konsequenz der Einhaltung bestimmter ungeschriebener Richtlinien machen. Zum Beispiel sind nicht alle Teile des Quellcodes ausreichend kommentiert und Variablen haben nicht immer aussagekräftige Namen. Desweiteren ist es nicht von Vorteil, bestimmte globale Variablen zum zentralen Bestandteil der Kommunikation zwischen den einzelnen Modulen zu machen, da dies das Maß an Übersichtlichkeit und vor allem an Verständlichkeit für den „Gdb-Neuling" verringert. Läßt man diese vergleichsweise kleine Zahl an Mängeln außer Acht, so können die C-Quellen mit einer Reihe von vorgefertigten Konzepten aufwarten. Es ist zum Beispiel möglich, der Gdb-Shell neue Kommandozeilen-Befehle hinzuzufügen. Wenn der Gdb gestartet wird, dann baut er sich eine Liste mit Befehlen auf. Man kann durch den Ruf der Funktion
add_cmd(<Befehl>,<Klasse>,<Funktion>,<Doku>,<Liste>)
den Kommandonamen <Befehl> mit der C-Funktion <Funktion> verknüpfen. Wenn man diesen Befehl in die globale Liste der Befehle einfügt, so ist er danach Bestandteil des Gdb. Weiterhin kann man hier noch einen kleinen Hilfetext in <Doku> angeben, der dem Nutzer die Anwendung des neuen Kommandos erläutert. Die Semantik des neuen Befehls kann nun in der Funktion <Funktion> implementiert werden. Da alle weiteren Standardkommandos (`break, next, continue` etc.) des Gdb auf die gleiche Weise beim Start des Gdb in die globale Liste der Befehle aufgenommen werden, sind sie auch intern durch Aufruf der entsprechenden mit ihnen verknüpften Funktionen benutzbar. Das ermöglicht ein Zusammensetzen von Befehlen auf Implementierungsebene. Beim Hinzufügen von neuen Basisbefehlen sollte immer darauf geachtet werden, daß diese für die Definition komplexer Kommandos verwendbar sind.

6.2 Änderungen an der Thread-Bibliothek

Die FSU-Pthreads sind eine Bibliotheksimplementierung, in der eine Reihe von Funktionen für die Arbeit mit Threads zur Verfügung gestellt werden. Man kann diese grob in folgende Klassen einteilen:

- Thread Management,
- Synchronisation und
- Scheduling.

Die einzelnen Vertreter der Klassen sollen an dieser Stelle nicht weiter beschrieben sein. Hierzu liefert die Schnittstellen-Beschreibung der FSU-Pthreads [7] ausreichend Informationen.
Es muß nun eine weitere Klasse, sogenannte **Debugging-Funktionen**, hinzugefügt werden. Obwohl der in Abschnitt 5.2 beschriebene Satz an neuen Befehlen für den Debugger eine große Zahl an Funktionen erwarten läßt, sind es zum gegenwärtigen Stand folgende :

- `int pthread_tid(int mode)`
  gibt die Liste aller Threads zurück und zusätzlich noch Statusinformationen.
- `void pthread_switch_th(pthread_t t)`
  ordnet Thread t am Anfang der Run-Queue ein.
- `void pthread_suspend_th(pthread_t t)`
  entfernt Thread t aus allen Queues.
- `void pthread_resume_th(pthread_t t)`
  ordnet Thread t am Ende der Run-Queue wieder ein.
- `get_next_pc(void)`
  gibt den *pc* des nächsten auszuführenden Threads zurück.

Die Funktion `pthread_tid()` dient dazu, die zentralen Datenstrukturen, in denen alle Thread-Informationen gehalten werden, auszulesen und in Variablen abzulegen. Dabei werden nicht alle Informationen gelesen, sondern nur die im Zusammenhang mit dem Debugging interessanten. Der Debugger hat Informationen über die Offsets aller Variablen aus der Symboltabelle, so daß er auf sie mittels `ptrace()` zugreifen kann. Auf Probleme, die bei der Implementierung entstanden sind, wird im Abschnitt 6.3 eingegangen. Die Funktionen `pthread_switch_th()`, `pthread_suspend_th()` und `pthread_resume_th()` arbeiten auf den Scheduling-Listen. Hierbei ist jedoch zu beachten, daß bei diesen Operationen keine Rücksicht auf den momentanen Zustand eines Threads genommen wird und daß Prioritäten nicht beachtet werden. Die im Zusammenhang mit diesen Funktionen stehenden Kommandos des Debugger sollten nur bei genauer Kenntnis der Funktion der einzelnen Threads verwendet werden.

6.3 info threads

Mit diesem Kommando soll dem Nutzer ein schneller Überblick über alle Threads und deren Status gegeben werden. Im Vorfeld der Implementierung dieses Kommandos ergaben sich folgende Fragen:

- Wie kann ein Datenaustausch zwischen Gdb und Pthreads erfolgen ?
- Wie geht man effizient mit den gewonnenen Daten um ?

Zuerst zur Frage des Datenaustauschs. Der Gdb bietet hier einen Mechanismus an, der es erlaubt, Ausdrücke der Sprache C zu parsen und auszuwerten. Dafür stehen die Funktionen

parse_and_eval(<C-Expression>)
und
parse_and_eval_address(<C-Expression>)

zur Verfügung.
Insbesondere ist es möglich, Funktionsaufrufe im C-Expression zu verwenden. Der Gdb schreibt bei einem Funktionsaufruf den dazugehörigen Aktivierungsrahmen auf den Stack des Ziel-Prozesses. Bevor dieser Code von der Applikation ausgeführt wird, werden noch alle zur Ausführung des ursprünglichen Programms notwendigen Daten gesichert. Nach dem Funktionsaufruf kann so der ursprüngliche Zustand wiederhergestellt werden.
Zugriff auf die Daten der Applikation kann jetzt realisiert werden, indem man eine Funktion der Thread-Implementierung ruft, die die internen Daten der Applikation liest und in Variablen ablegt, die dann wiederum gelesen werden können. Nun kann man sich fragen, warum erst ein Aufruf und dann noch der Aufwand für das Lesen der geschriebenen Variablen durchgeführt werden. Man könnte natürlich auch direkt mit Hilfe der oben erwähnten Funktionen auf die Daten zugreifen. Man hat so aber den Vorteil, daß die

Debug-Funktion in der Thread-Implementierung vom Entwickler selbst geschrieben werden kann und komplizierte Zugriffe auf interne Daten verborgen bleiben. Alle relevanten Informationen über die Threads werden dagegen auf sehr elegante Weise in vorher festgelegte Variablen geschrieben, deren Name und Typ bekannt sind. Wird die Thread-Implementierung geändert, d.h. ohne die Namen und Typen der für die Kommunikation wichtigen Variablen zu ändern, so können die Zugriffe auf diese Variablen auf der Seite des Gdb beibehalten werden.
Bei der Umsetzung dieses Verfahrens ergab sich auch noch das folgende Problem. Die Daten, die von der Thread-Implemetierung zur Verfügung gestellt werden, sollten immer atomare Typen (*float, integer, char* usw.) haben, sofern bei der Kompilierung des Quelltextes der Thread-Implementierung Optimierungen gemacht werden, die Einfluß auf die Speicherung von komplexen Datenstrukturen haben. Verwendet man Arrays oder andere komplexe Datentypen, so besteht die Gefahr, daß diese nicht zusammenhängend, sondern fragmentiert abgespeichert sind. Will man mit dem Gdb auf einzelne Elemente zugreifen, so kann er die Position im Speicher nicht erkennen, da er nicht im Besitz der Allozierungstabellen ist. Deswegen sollte man alle Ergebnisse mit Hilfe von Variablen atomaren Typs übergeben.

6.4 continue -cs

Da eine Ausführung von n Threads auf einer Maschine mit p Prozessoren ($p < n$) ein Scheduling notwendig macht, kommt es während des Applikationslaufes zu Kontextwechseln. Um die Arbeitsweise des Schedulers zu beobachten, ist es von Vorteil, die Kontextwechsel zu betrachten. Von besonderem Interesse ist dabei der Zeitpunkt des nächsten Kontextwechsels und welche Threads daran beteiligt sind.
Diesem Zweck soll eine zusätzliche Option (**-cs** für „context switch") für den **continue**-Befehl dienen. Bei der Implementierung der neuen Funktionalität gab es zwei Ansätze.
Im ersten Ansatz wurde eine neue Funktion `f()` in die Bibliothek eingeführt, die keine Funktionalität hatte, sondern nur am Ende der Schedulingfunktion gerufen wurde. Um während eines **continue** direkt nach einem Kontextwechsel zu stoppen, war es nur notwendig, vor dem Ruf des **continue**-Befehls einen Breakpoint B_1 bei der Schedulingfunktion, hier `sched()` genannt, und einen Breakpoint B_2 bei `f()` zu setzen. Da der Nutzer jedoch nicht bei B_1, sondern erst bei B_2 (nach dem Kontextwechsel) stoppen sollte,

wurde eine Eigenschaft der Breakpoints des Gdb genutzt. Es ist möglich, mit einem Breakpoint eine Folge von weiteren Gdb-Kommandos zu assoziieren, die bei Erreichen des Breakpoints ausgeführt werden. Um nun das Problem des Stoppens bei B_1 zu umgehen, war es notwendig, B_1 und B_2 mit folgenden Kommandos zu verbinden.

- Kommandos für B_1:
  echo "context switch:"
  refresh_thread_table
  print_current_thread_number
  echo "→"
  disable B_1
  continue

- Kommandos für B_2:
  print_current_thread_number
  echo " \n"
  disable B_2

Der Effekt dieser Kommandos ist die Ausgabe der aktuellen Thread-Nummer vor der Ausführung der Schedulingfunktion und ein **continue** zum nächsten Breakpoint, bei dessen Erreichen dann wiederum die aktuelle Thread-Nummer ausgeschrieben und zur Gdb-Shell zurückgekehrt wird. Der Nutzer sieht dann zum Beispiel folgende Ausschrift des Debuggers:

```
Breakpoint 1, slave (num=0x80571f4) at ms_mutex.c:22
22        int a,i, n = rand() % 1000;
```

(gdb) **continue -cs**

```
Continuing.
context switch:
T 2 (805b400)-->T 1 (806a000)
(gdb)
```

Es gibt jedoch Schwachstellen dieses Ansatzes. Die Funktion `f()` wird immer in `sched()` gerufen. Die Änderungen an der Thread-Implementierung

sollen aber keine Auswirkungen auf nicht zu debuggende Applikationen haben. Weiterhin kann es auch Pthreads-Implementierungen geben, in denen ein einfaches Einfügen dieses Funktionsrufes nicht möglich ist, da die Scheduling-Routinen nicht im Quellformat vorliegen. Eine weitere Schwachstelle ist die Tatsache, daß wir uns bei Erreichen von B_2 eigentlich noch in Bibliothekscode befinden. Der Gdb soll sich jedoch bei der ersten nach dem Kontextwechsel ausgeführten Zeile des Threads befinden. Das ermöglicht nun der zweite Ansatz.
Die Bibliothek hält zu jedem Thread eine Reihe von Informationen. Unter anderem ist auch bekannt, welcher Thread nach dem aktuellen ausgeführt wird, und aus dem gespeicherten Kontext ist ersichtlich, an welchem *pc* die nächste ausgeführte Instruktion steht. Setzt man nun B_2 an diesem *pc* (dies ist Gdb-intern möglich), so kann man den Funktionsruf von `f()` sparen. Zu diesem Zweck muß uns die Bibliothek jedoch einen eleganten Zugriff auf den *pc* des nächsten auszuführenden Threads geben. Dazu ist eine Funktion `get_next_pc()` in die Bibliothek eingeführt worden. Die an die Breakpoints angehängten Kommandos können in etwa übernommen werden. Breakpoint B_1 kann statisch gesetzt werden, jedoch muß B_2 bei jedem Aufruf von **continue -cs** vom Gdb neu gesetzt werden, nämlich auf den von `get_next_pc()` ermittelten *pc*.

6.5 nextth

Dieses Kommando soll es ermöglichen, die nächste ausgeführte Zeile des aktuellen Threads zu erreichen. Der herkömmliche Weg des Gdb für sequentielle Programme ist es, das Programm Maschineninstruktionen ausführen zu lassen, bis der letzte *pc* (program counter) der aktuellen C-Zeile erreicht ist. Jetzt wird noch eine Einzelinstruktion ausgeführt. Der *pc* zeigt dann auf den Anfang der nächsten ausgeführten C-Zeile.
Diese Vorgehensweise wurde im Hinblick auf ein nebenläufiges Programm abgewandelt. Der Gdb soll das Programm jetzt auch Einzelinstruktionen ausführen lassen, aber bis zum Erreichen der nächsten ausgeführten Zeile des aktuellen Threads. Der Pseudo-Code für dieses Kommando sieht folgendermaßen aus:

```
function target_pc()
  begin
    return aktueller pc des Target-Prozesses
```

```
  end

function Zeile(pc)
  begin
    return Nummer der mit pc assoziierten Quellzeile
  end

procedure nextth()
  begin
    T:=aktueller_Thread()
    Z:=Zeile (target_pc())
    while not (aktueller_Thread()=T) or (Zeile(target_pc())=Z)
     do
      lasse das Programm eine Instruktion durchführen
     od
  end
```

Für den Fall, daß ein Kontextwechsel nach der Ausführung der aktuellen Zeile stattfindet, gibt es zwei Phasen. In der ersten Phase, in der sich das Programm noch in der aktuellen Zeile im Thread T befindet, ist die zweite Bedingung $Cond_2 := Zeile(target\_pc()) = Z$ erfüllt. Wird jetzt ein Kontextwechsel gemacht, so ist die Bedingung $Cond_1 := not(aktueller\_Thread() = T)$ erfüllt. Das Kommando nextth stoppt also nur, wenn T der aktuelle Thread ist und der pc in der nächsten von Z verschiedenen Zeile liegt. Es ist damit auch der Fall eingeschlossen, daß kein Kontextwechsel stattfindet. Dann gibt es nur die Phase, in der $Cond_2$ erfüllt ist. Danach befinden wir uns in Thread T, da kein Kontextwechsel stattfand und die neue Zeile ungleich Z ist.

6.6 breakth

Dieser Befehl soll es ermöglichen, threadspezifische Breakpoints zu setzen. Aus verschiedenen Gesichtspunkten können diese nützlich sein:

- zur Prüfung, ob eine Stelle des Quelltextes von einem bestimmten Thread erreicht wird, und
- zum Ausblenden der Aktionen anderer Threads und damit dem Erreichen von mehr Übersichtlichkeit.

Der **breakth**-Befehl sollte benutzbar sein, wie der im Gdb bereits vorhandene **break**-Befehl, d.h man kann als Stopp-Lokalität alle Möglichkeiten des **break**-Befehls nutzen (Stopp bei Zeilennummern, Funktionen und *pc*). Als neue Funktonalität kommt hinzu, daß man zusätzlich die Threads spezifizieren kann, bei denen eine Unterbrechung stattfindet. Das geschieht durch Verwendung des neuen Befehls **breakth** und die zusätzliche Angabe der Thread-Nummer. Es wird somit die neue Klasse der threadspezifischen Breakpoints eingeführt. Damit entsteht ein Konflikt zwischen herkömmlichen und threadspezifischen Breakpoints. Was passiert, wenn ein threadspezifischer Breakpoint und ein herkömmlicher Breakpoint gleich spezifiziert werden, z.B.:

- `break f()`
- `breakth 2 f()`.

Dieser Konflikt wurde dadurch gelöst, daß der zuerst spezifizierte Breakpoint Vorrang hat. Will man also im Beispiel nach dem ersten **break** noch ein **breakth** an der gleichen Stelle für Thread Nummer 2 durchführen, so muß man den durch **break** erzeugten Breakpoint löschen. Als oberstes Prüfkriterium für die Implementierung und als Maßgabe für den Lösungsansatz galten die im Gdb verwirklichten Befehle. Insbesondere gab es ein Problem beim **breakth** zu lösen. In der ersten Implementierung des neuen Befehls war es nicht möglich, einen threadspezifischen Breakpoint vor dem Start des Programms zu setzen. Das ist jedoch beim **break**-Befehl möglich. Auf dieses und andere Probleme und deren Lösung wird später genauer eingegangen. Zuvor soll die grundsätzliche Lösung vorgestellt werden. Im Gdb sind folgende wichtige Datenstrukturen und Funktionen vorhanden, die unmittelbar mit Breakpoints verbunden sind. Die zentrale Struktur zur Speicherung von Breakpoint-Daten ist `struct breakpoint`, die in der Gdb-Quelldatei breakpoint.h definiert wird. Hier ein Ausschnitt:

```
struct breakpoint
{
  struct breakpoint *next;
  enum bptype type; /* Type of breakpoint. */
  enum enable enable; /* Zero means disabled ...   */
  enum bpdisp disposition;
  int number;
  CORE_ADDR address; /* Address to break at ...  */
```

```
  int line_number;
  char *source_file;
  unsigned char silent; /* Non-zero means silent ... */
  int ignore_count;
  ...
  struct command_line *commands; /* ... execute when hit this bp  */
  CORE_ADDR frame;
  ...
  char *addr_string;
  ...
  int thread; /* Thread number .... */
  int hit_count;

};
```

An der Anzahl der in der Struktur vorhandenen Einträge ist schon erkennbar, welche Möglichkeiten man bei der Spezifikation eines Breakpoints hat. Bei der Behandlung eines Breakpoints sind folgende Funktionen beteiligt. Setzt man einen Breakpoint mittels break <Lokalität>, so wird die Funktion `break_command()` aus *breakpoint.c* mit dem Argument <Lokalität> aufgerufen. Es wird eine Struktur vom Typ `struct breakpoint` erzeugt und mit den wichtigsten Informationen gefüllt. Desweiteren wird im zu debuggenden Programm am errechneten *pc* eine Trap-Instruktion eingefügt. Wird im Laufe der Programmabarbeitung diese Instruktion erreicht, so geht die Kontrolle wieder an den Gdb über. Dieser führt jetzt einen Vergleich verschiedener Zustandsdaten des Programms mit den in der Struktur spezifizierten Daten durch. Unter anderem wird auch geprüft, ob die angegebene Thread-Nummer mit der aktuellen Nummer übereinstimmt. Im Falle der Übereinstimmung wird die Kontrolle an den Benutzer übergeben, andernfalls wird das Programm durch den Gdb fortgeführt. Dieser Vergleichsalgorithmus war bereits im Gdb implementiert. Die Voraussetzungen für das Setzen threadspezifischer Breakpoints waren somit sehr gut. Um eine Lösung für Programme zu erreichen, die die Pthreads nutzen, ergaben sich folgende Punkte, die es zu lösen galt:

1. Eintragen der Thread-Nummer in die Struktur des bereits mit `break_command()` erzeugten Breakpoints

2. Neuimplementierung des Vergleichs der Thread-Nummern des aktuellen und des angegebenen Threads.

6.6.1 Eintragen der Threadnummer

Das Eintragen der Thread-Nummer in die durch `break_command()` erzeugte und mit den wichtigsten Daten gefüllte Struktur geschieht folgendermaßen. Zuerst muß man die Struktur aus der globalen Liste der Breakpoints heraussuchen, da `break_command()` keine Breakpoint-Nummern zurückgibt. Da mehrere Breakpoints an dieser Stelle gesetzt sein können, sucht man nach der Struktur mit der höchsten Thread-Nummer, denn das ist die zuletzt erzeugte Struktur. In diese trägt man die Nummer des Threads ein. Hier ergab sich folgendes Problem. In der Thread-Bibliothek waren keine Informationen über die Nummern der erzeugten Threads verfügbar. Lediglich die Adressen der für die Speicherung der Thread-Informationen angelegten Strukturen waren verfügbar. Das machte es erforderlich, eine Numerierung auf der Gdb-Seite durchzuführen. Dazu wird die bereits erläuterte Funktion `pthread_tid()` ausgeführt, und es werden durch den Gdb alle Thread-Daten ausgelesen. Jetzt erfolgt eine Numerieung der Threads. Das ist einerseits problematisch, da die Nummern während des Programmlaufs eindeutig sein müssen, d.h. wird ein Thread gelöscht, so darf seine Nummer bei der nächsten Numerierung nicht mehr verwendet werden. Hierzu ist dann eine Vergleichstabelle erforderlich, die einer Struktur-Adresse eine Thread-Nummer zuordnet. Ein anderes Problem ist, daß das Setzen eines threadspezifischen Breakpoints nur zur Laufzeit des Programms möglich ist, da die erzeugten Threads numeriert werden müssen. Es kann außerdem sein, daß zum Zeitpunkt des Setzens des Breakpoints der spezifizierte Thread noch nicht erzeugt wurde. Diese Probleme können gelöst werden, indem man in die Thread-Struktur der Bibliothek einen Eintrag für die Thread-Nummer einfügt. Darum soll hier generell gefordert werden, daß alle Thread-Implementierungen eine solche Information bereitstellen. Die Numerierung der Threads auf Bibliotheksseite kann bei der Erzeugung der Threads erfolgen. Bei den Pthreads ist dafür die Funktion `pthread_create()` zuständig, hier ist mit geringem Aufwand das Eintragen der Thread-Nummer zu realisieren. Es ist nach der Änderung der Bibliothek nur erforderlich, die Nummer des spezifizierten Threads in die Struktur des Breakpoints einzutragen. Da zur Laufzeit des Programms die Nummer des aktuellen Threads vorliegt, ist diese ohne Probleme mit der spezifizierten Thread-Nummer vergleichbar.

6.6.2 Vergleich der Threadnummern

Der Vergleich der Thread-Nummern gestaltet sich nach der eben angesprochenen Änderung der Bibliothek sehr einfach. In der vom Gdb für den Vergleich der Thread-Nummern aufgerufenen Funktion `breakpoint_thread_match()` wird ein Lesen der aktuellen Thread-Nummer durchgeführt. Dazu wird dieses Datum aus dem Prozeß-Abbild des zu debuggenden Programms gelesen. Diese wird dann mit der spezifizierten Nummer verglichen, und die Funktion kehrt mit dem Ergebnis des Vergleichs zurück. Stimmt die Thread-Nummer überein, so bekommt der Nutzer die Kontrolle über den Gdb zurück, andernfalls wird das Programm weitergeführt.

6.7 switchth, suspendth, resumeth

Mit diesen Befehlen soll man auf das Scheduling der einzelnen Threads Einfluß nehmen können. Die Kommandos **switchth**, **suspendth** und **resumeth** setzen sich durch ihren Charakter von den in Abschnitt 5.2 vorgeschlagenen Kommandos ab. Sie haben direkten Einfluß auf die internen Strukturen der Thread-Implementierung und sind speziell auf diese ausgerichtet.
Die bisher noch nicht vollständig implementierte Lösung hat folgenden Ansatz. Die gesamte Funktionalität der Schedulingbefehle wird sich in der Thread-Implementierung befinden. Nur die Funktionen selbst werden vom Gdb gerufen. Da **switchth**, **suspendth** und **resumeth** auf den Thread-Warteschlangen arbeiten und die damit verbundenen Informationen von den Bibliotheksfunktionen verwaltet werden, sollen die an der Bibliothek erforderlichen Änderungen hier nicht näher behandelt werden.

7 Architekturspezifische Details und Änderungen

Der Gdb ist ein Werkzeug, daß auf vielen Architekturen lauffähig ist. Deshalb war das Bestreben groß, auch die Änderungen in Bezug auf das Thread-Debugging für die unterstützten Architekturen verfügbar zu machen. Eine Reihe von Problemen ergab sich.
Auf der SUN Sparc war folgendes zu beobachten. Diese Maschine unterstützt kein „single step", welches vom Gdb ausgenutzt wird, um **stepi** auszuführen. Dafür wurde in den architekturspezifischen C-Quellen des Gdb eine Softwa-

reemulation des Hardware-„single step" angeboten. Diese mußte jedoch in geeigneter Weise und an den richtigen Stellen vom Gdb benutzt werden. Die Emulation kann man durch wiederholtes Setzen von Breakpoints und dem damit verbundenen Einfügen von Trap-Instruktionen im Kind-Prozeß erreichen. Dabei wird wie in Abbildung 2 verfahren.

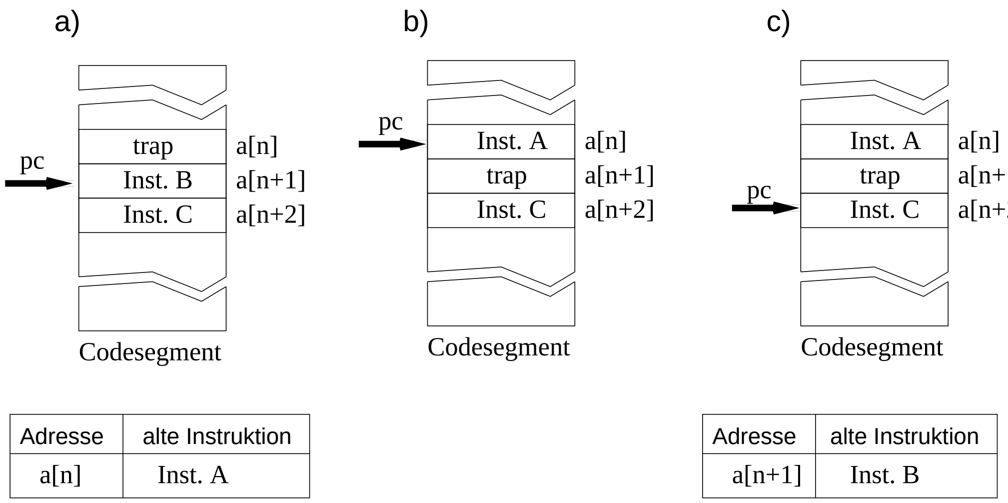

Abbildung 2: *Nachbilden des Hardware-„single step" durch Verwendung von Trap-Instruktionen*

In Bild a) ist die Anfangssituation dargestellt. Bei Adresse a[n] wurde mittels Ersetzung der alten Instruktion durch eine Trap-Instruktion ein Breakpoint gesetzt. Diese Trap-Instruktion wurde als letztes ausgeführt und die Kontrolle ging somit auf den Gdb über. Der *pc* des Programms zeigt jetzt auf a[n+1]. Die Instruktion A, die ursprünglich in a[n] gespeichert war, ist bisher noch nicht ausgeführt worden. Wird an dieser Stelle ein „single step" ausgeführt, so bedeutet das die Ausführung der Instruktion A. Dazu wird, wie in Bild b) dargestellt, die alte Instruktion nach a[n] geschrieben, und bei a[n+1] ein „trap" eingefügt [2], um nach genau einer ausgeführten Instruktion zu stoppen. Der *pc* wird wieder auf a[n] gesetzt, um die Ausführung der Instruktion A zu beginnen. Der nächste Stopp des Programmes endet in dem

[2] Bei Kontrollflußänderungen muß die Adresse der nächsten ausgeführten Instruktion (das Sprungziel) ermittelt werden, indem die Sprungbedingung abgetestet wird.

durch Bild c) beschriebenen Zustand. Instruktion A wurde ausgeführt und nach der Trap-Instruktion ging die Kontrolle wieder an den Gdb zurück. Das entspricht der Semantik eines „single step". Für ein weiteres „single step" wird so verfahren, als befände man sich im Zustand a).

Desweiteren gab es mit der SUN Sparc ein Problem mit dem Zurückverfolgen der Stack-Informationen. Folgende Beobachtungen waren zu machen. Nach dem Erreichen eines Breakpoints konnte der Gdb keine Stack-Informationen bereitstellen. Bei einem nachträglich ausgeführten **jump** auf das Ziel des Breakpoints wurden jedoch alle Stack-Informationen erhalten. Die Ursache dieses Verhaltens konnte bisher nicht geklärt werden. Das Anfügen eines **jump** an jeden gesetzten Breakpopint ist aber eine praktikable Lösung. Hierbei ist nur die Ausführung eines zusätzlichen `ptrace()`-Aufrufes nötig.

Bei der Linux-Version des Gdb kam es nach der Einführung eines neuen Kernels zu einer erheblichen Verlängerung der Ausführungszeit des Kommandos **info threads**. Entscheidenden Einfluß darauf hatte der Systemruf `ptrace()`. Deshalb war es notwendig, eine andere Form des Datenaustauschs zwischen Gdb und Pthreads zu wählen.

Folgende Lösung wurde gewählt. Es wird die Möglichkeit des gemeinsamen Speichers („shared memory") benutzt, die in allen UNIX-System-V-Derivaten vorhanden ist. Der in Abschnitt 6.3 erklärte Ansatz ändert sich dann wie folgt. Es wird wiederum eine Funktion der Thread-Implementierung gerufen. Diese schreibt dann in den vorher vom Gdb installierten gemeinsamen Speicher, dessen Schlüssel die Prozeßnummer des gerade ausgeführten Gdb ist. Danach kann der gemeinsame Speicher vom Gdb ausgelesen werden (siehe Abb. 3).

Ein Nachteil dieser Lösung ist, daß in den Quellen des Gdb und der Pthreads die gleichen Header-Dateien mit Informationen über die in den gemeinsamen Speicher geschriebenen Datenstrukturen vorhanden sein müssen. Dafür entschädigt jedoch der Geschwindigkeitsvorteil, den man durch die Benutzung des Shared Memory erlangt. Man benötigt für die Ausführung von **info threads** nur noch einen Funktionsaufruf, den der Gdb im Kind-Prozeß initiieren muß. Es gibt keine weiteren Zugriffe auf den Speicher der zu debuggenden Applikation, d.h. keine weiteren Aufrufe von `ptrace()`.

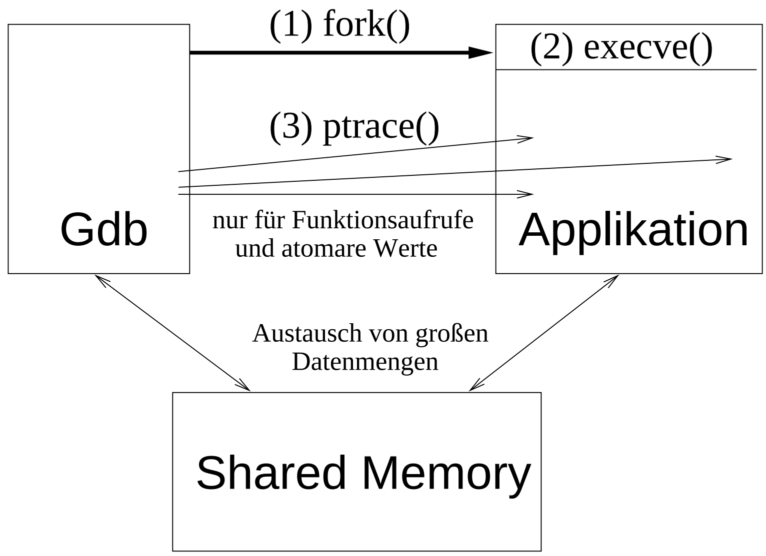

Abbildung 3: *Datenaustausch mit Shared Memory*

8 Verwandte Forschungsarbeiten

McDowell und Helmbold [5] geben eine generelle Einführung in die Thematik des Debuggings nebenläufiger und verteilter Programme. Hier werden Breakpoint-Debugging, Event-basiertes Debugging sowie die statische Analyse und deren Anwendung auf nebenläufige und verteilte Anwendungen diskutiert.
Einen Debugger für Thread-Anwendungen stellen Caswell und Black [2] vor. Dieser Debugger ist eine Änderung des Gdb und stellt eine Lösung für das Betriebssystem Mach dar, in dem Threads bereits vom Betriebssystemkern unterstützt werden. Die Änderungen von Caswell und Black boten zugleich Anregungen und Anlaß zur Kritik. Die vorgestellte Implementierung diente als Anhaltspunkt für das Finden der Quelldateien, in denen Änderungen am besten zu plazieren waren. Generell half sie aber auch beim Verständnis der Abläufe im Gdb. Die hinzugefügten Befehle waren

- *threads* zur Ausgabe von Thread-Informationen,
- *thread* $<$*Nummer* $>$ für das Wechseln zu einem bestimmten Thread und

- *apply* <*Threadliste* ><*Kommando* > zum Anwenden eines Kommandos auf eine Liste von Threads.

Dieser Satz an Befehlen hat noch nicht die Funktionalität der in Abschnitt 5.2 geforderten Basisbefehle. Kritikwürdig ist unter anderem, daß ein Befehl zum Setzen von threadspezifischen Breakpoints fehlt.
Ponamgi, Hseush und Kaiser [8] stellen eine Arbeit vor, die sich mit dem Erkennen von Event-Mustern durch den von Caswell und Black erweiterten Debugger beschäftigt. Hier werden 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 Sprache der DPE's 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.
In einer sehr umfangreichen Erweiterung des Gdb, dem *SmartGDB* von Halbhavi [4], ist ebenfalls eine zusätzliche Programmierschnittstelle eingeführt worden. Ein *Tcl*-Interpreter wurde so integriert, daß die Befehle beider Werkzeuge, des Gdb und des *Tcl*-Interpreter, in einer Kommandooberfläche benutzbar sind. Mit dieser Architektur werden den Anwendungen die Fähigkeiten der jeweils anderen übertragen. Nachteilig ist jedoch, daß Namenskonflikte durch Umbenennung der Gdb-Kommandos gelöst wurden. Um es dem Nutzer zu ermöglichen, komplexe Befehle für ein Thread-Debugging zu definieren, mußten dem Gdb jedoch einige Basisbefehle hinzugefügt werden. Diese sind vergleichbar mit den in 5.2 angesprochenen Befehlen **info threads**, **switchth** und **breakth**. Die beiden letztgenannten Befehle wurden schon mit Hilfe von *Tcl* implementiert. Desweiteren wurde ein Kommando zum Ausgeben des Stacks jedes einzelnen Threads vorgestellt. In der hier vorgestellten Arbeit ist es nur möglich, den Stack des aktuellen Threads zu betrachten.
Anhand der erwähnten Arbeiten ist deutlich zu erkennen, daß der Gdb eine solide Basis für die Entwicklung eines Thread-Debuggers ist. Die Erweiterungen in [8, 4] zielen auf eine bessere Anpassung des Gdb an die individuellen Anforderungen des Nutzers ab. Das Programmieren von speziellen Befehlen ist jedoch sehr zeitintensiv und durch die zusätzliche Einbindung eines Inter-

preters nimmt die Größe des Gdb zu. Leistungsfähige Basisbefehle, wie sie hier vorgestellt wurden, haben ihre Berechtigung, da sie einerseits Grundlage für die Konstruktion komplexer Befehle sind und andererseits ein schnelles Arbeiten ohne vorherige Programmierung ermöglichen.

9 Ausblick

Es wurden bisher einige Änderungen am Gdb vollzogen, die aber momentan nur eine sehr eingeschränkte Lösung darstellen. So wird derzeit eine Debug-Unterstützung ausschließlich für die FSU-Pthreads angeboten. Zukünftig sollte eine einheitliche Lösung für eine große Anzahl von Thread-Implementierungen erarbeitet werden. Die grundsätzliche Idee dafür sieht folgendermaßen aus. Es wird eine einheitliche Schnittstelle geben, über die der Gdb mit der Thread-Implementierung kommuniziert (siehe Abb. 4). Die Anzahl der Funk-

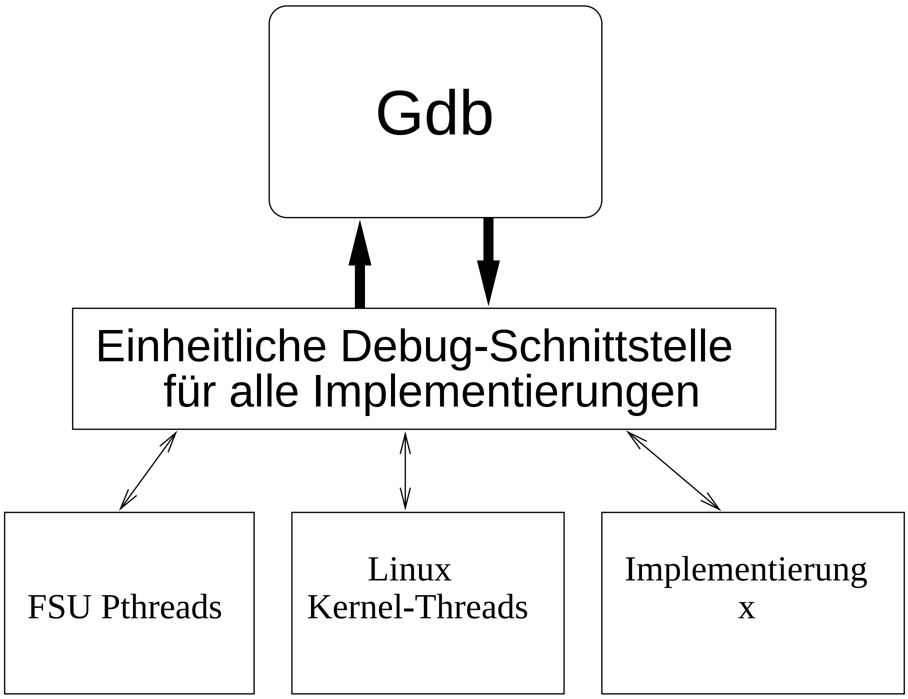

Abbildung 4: *Eine einheitliche Schnittstelle*

tionen, die von jeder Thread-Implementierung bereitgestellt werden müßten, ist überschaubar klein, so daß die Implementierung einen geringen Aufwand darstellt. Die Aufrufparameter der Funktionen, Funktionsnamen, Rückgabewerte und Typen sollten vereinheitlicht werden, so daß der Gdb ohne gravierende Änderungen im Quelltext für jede Bibliotheks- und Kernelimplementierung neu eingerichtet werden könnte. Die Semantik und alle internen Berechnungen würden auf der Bibliotheks- bzw. Kernelseite implementiert werden und für den Gdb verborgen sein. Weitere Punkte einer zukünftigen Arbeit sind

- die vollständige Implementierung der Befehle **switchth**, **suspendth** und **resumeth**,
- das Anzeigen von Stacks anderer Threads sowie
- die Änderungen an den *Makefiles* zur automatischen Konfiguration des Gdb für die von den Pthreads unterstützten Architekturen.

Weiterhin ist es naheliegend, eine Debug-Unterstützung für verteilte Thread-basierte Anwendungen zu entwickeln.

10 Schlußfolgerungen

Motivation für diese Arbeit war es, einen Debugger zu schaffen, der es ermöglicht, Thread-Applikationen in einer für sie geeigneten Weise testen und untersuchen zu können. Existierende Debugger boten diese Möglichkeit bisher nicht oder nur unzureichend.

Im Laufe der Arbeit wurden grundlegende Fragen im Zusammehang mit dem Debugging nebenläufiger Anwendungen beantwortet und erste Lösungen erarbeitet. Ergebnis ist eine obligatorische Menge von Befehlen, die ein Debugger anbieten sollte, um ein Thread-Debugging zu ermöglichen. Eine praktische Realisierung einer Teilmenge dieser Befehle erfolgte mit der Modifizierung des Debuggers Gdb sowie der Anpassung einer bestehenden POSIX-Threads-Implementierung, der FSU-Pthreads-Bibliothek.

Mit der Implementierung ist es nun möglich, Informationen über alle gestarteten Threads zu erhalten, threadspezifische Breakpoints zu setzen und Kontextwechsel zu erkennen und zu kontrollieren. Weiterhin wurden Befehle hinzugefügt, die noch experimentellen Charakter haben. Hierzu gehören solche, mit denen der Nutzer in das Scheduling der Threads eingreifen kann und ein Befehl, mit dessen Hilfe man die nächste Quelltextzeile erreicht, ohne den

aktuellen Thread zu verlassen.
Die prototypische Lösung diente dazu, Machbarkeit und Aufwand für diese Aufgabe festzustellen. Im Hinblick auf die Erarbeitung einer einheitlichen Schnittstelle zwischen einem Debugger und einer beliebigen Thread-Implementation konnten erste Erfahrungen und Lösungsansätze gesammelt werden. Die Anforderungen an Debugger und Thread-Implementation wurden herausgearbeitet und können als Grundlage für die Anpassung verschiedener Debugger und Thread-Implementationen verwendet werden. Eine anschließende Diplomarbeit wird sich eingehender mit der Frage einer einheitlichen Schnittstelle beschäftigen.

Literatur

[1] Helmut Balzert. *Lehrbuch der Software-Technik.* Heidelberg, 1996.

[2] Deborah Caswell and David Black. Implementing a mach debugger for multithreaded applications. In *USENIX*, pages 25–40, Winter 1990.

[3] Linux FAQ. *Kernel Threads und GDB?* http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html, 1997.

[4] Sudhir Halbhavi. *Thread Debugger Implementation and Integration with the SmartGDB Debugging Paradigm.* University of Kansas, 1996.

[5] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. Technical Report UCSC-CRL-88-21, University of California, Santa Cruz, Computer Research Laboratory, November 1988.

[6] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[7] Frank Mueller. *Pthreads Library Interface.* Florida State University, Tallahassee, 1994.

[8] M. Krish Ponamgi, Wenwey Hseush, and Gail E. Kaiser. Debugging multi-threaded programs with MpD. *IEEE Software*, May 1991.

Anhang: Für die Debug-Unterstützung erforderliche Funktionen der Thread-Implementation und des Debuggers

Da die benutzte Thread-Implementierung und der Gdb in C geschrieben sind, werden im folgenden die C-Signaturen der Funktionen angegeben.

Erforderliche Funktionen[3] der FSU-Pthreads:

- `int next_pc()` - gibt den Wert des pc zurück, bei dem der nächste Thread seine Ausführung fortsetzt
- `int pthread_tid()` - gibt die ID des aktuellen Threads zurück und stellt die Liste aller Threads in einer für den Debugger lesbaren Form (z.B. „shared memory" oder globale Variable) bereit
- `void pthread_switch_th(pthread_t t)` - veranlaßt einen Wechsel zum Kontext des Threads t
- `void pthread_suspend_th(pthread_t t)` - stoppt Thread t, sofern dieser aktiv ist und entfernt ihn aus allen Scheduling-Listen
- `void pthread_resume_th(pthread_t t)` - fügt Thread t wieder in die Schedulinglisten ein

Benutzte Funktionen des Gdb:

- `CORE_ADDR parse_and_eval_address(char *exp)` - wertet den in der Zeichenkette `exp` übergebenen C-Ausdruck aus und gibt das Ergebnis als Ganzzahlwert zurück
- `value_ptr parse_and_eval(char *exp)` - wertet den in der Zeichenkette `exp` angegebenen Ausdruck aus und legt das Ergebnis typgerecht in einer `struct value`-Struktur mit der Adresse `value_ptr` ab

[3] Alle aufgelisteten erforderlichen Funktionen der Thread-Bibliothek wurden neu implementiert.