Ronald Böhle

Refactoring einer nativen C++ Bibliothek(Syslib) mit dem Ziel der Bereitstellung der Funktionalität für andere Sprachen (Java) unter Berücksichtigung der Multithreading-Fähigkeit

eingereicht als

Diplomarbeit

an der

HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

Informatik

Mittweida, 2009

Erstprüfer: Prof. Dr. -Ing. Jürgen Ruck

Zweitprüfer: Thomas Klink

Vorgelegte Arbeit wurde verteidigt am:

Bibliographische Beschreibung:

Ronald Böhle:

Refactoring einer nativen C++ Bibliothek(Syslib) mit dem Ziel der Bereitstellung der Funktionalität für andere Sprachen (Java) unter Berücksichtigung der Multithreading-Fähigkeit. - 2009. 64 S.

Mittweida, Hochschule Mittweida (FH), Fachbereich Mathematik/Physik/Informatik, Diplomarbeit, 2009

Referat:

Ziel dieser Diplomarbeit ist es, eine C++-Anwendungsbibliothek threadsicher zu gestalten. Dies ist zum einen notwendig, da die Anwendungsbibliothek in anderen multithread-fähige Sprachen bereitgestellt wird. Zum anderen ist der Leistungsaspekt zu beachten.

Zu Beginn wird der aktuelle Aufbau der Anwendungsbibliothek analysiert. Danach werden Ansätze zur threadsicheren Gestaltung aufgezeigt und umgesetzt. Zum Abschluss der Diplomarbeit werden verschiedene Leistungstests durchgeführt und ausgewertet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

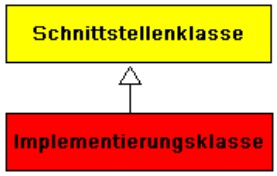
Konventionen

In diesem Abschnitt werden Konventionen und Normen vorgestellt, die in der Diplomarbeit verwendet werden. Die Konventionen der Datei- und Klassennamen sind Vorgaben von SAP.

Klassendiagramm

Die Klassendiagramme, die in dieser Diplomarbeit verwendet werden, sind farbig unterlegt. Folgende Tabelle zeigt die Bedeutung der Farben:

Klassen	Darstellung
Schnittstelle Interne Schnittstelle	gelb eingefärbt
Brücke Implementierung	rot eingefärbt



Darstellung der Klassendiagramme

Namenskonvention für Klassen

Klassennamen setzen sich aus verschiedenen Abkürzungen zusammen.
Beispiel:

ISyFSPathInt - [I][Sy][FSPath][Int] - [Schicht][Projekt][Name][Endung]

Die verwendeten Abkürzungen für Klassennamen sind in der folgenden Tabelle zusammengefasst:

Aufbau	Abkürzung	Beschreibung
Schicht	I	Schnittstelle / Interne Schnittstelle
	C	Brückenklasse / Implementierungsklasse
	P	Intelligenter Zeiger / Interner intelligenter Zeiger
Projekt	Sy	Syslib
	Thr	ThreadLib
Name	beliebig	Klassenname
Endung	<keine>	Schnittstelle / Brückenklasse / intelligenter Zeiger
	Int	Interne Schnittstelle / Interner intelligenter Zeiger
	Impl	Implementierungsklasse

Namenskonventionen für Klassennamen

In der Entwicklungsumgebung von SAP wird folgende Klasse verwendet:

- *iastring* - Eine Klasse für Zeichenketten, die auf der STL[1]-Klasse *wstring* aufgebaut ist und aus *wide character* besteht.

[1]Standard Template Library - Ein standardisiertes C++ Template Archiv für verschiedene Container-Klassen und Algorithmen

Namenskonvention für Dateien

Die Dateinamen der Quelltext-Dateien setzen sich aus verschiedenen Abkürzungen zusammen.
Beispiel:

iaxxithr.hpp - [ia][xx][i][thr][.hpp] - [Projekt][Plattform][Schicht][Name][Endung]

Die verwendeten Abkürzungen für Dateinamen sind in der folgenden Tabelle zusammengefasst:

Aufbau	Abkürzung	Beschreibung
Projekt	sy	Syslib
	ia	allgemeines Projekt
Plattform	xx	plattformübergreifend
	nx	Windows
	ux	Unix
Schicht	i	Schnittstelle
	b	Brückenklasse
	c	Implementierungsklasse
Name	beliebig	Bezeichnung / Abkürzung der Klasse
Endung	.hpp	Header-Datei
	.cpp	Quelltext-Datei

Namenskonventionen für Dateinamen

0. Einführung

0.1. Motivation

Eine neue Anforderung an moderne Software ist Parallelität. Diese ermöglicht eine parallele Bearbeitung verschiedener Aufgaben. Ein Grund für diese neue Anforderung sind fortschrittliche Prozessoren. Um die Leistung dieser Prozessoren zu steigern, wird in erster Linie nicht mehr die Leistung der einzelnen Prozessoren erhöht, sondern deren Anzahl. Doch diese neue Multiprozessor-Architektur allein reicht nicht aus, um die Leistungsfähigkeit von Software zu steigern. Sequenziell programmierte Software würde höchstens einen Prozessor belegen. Alle übrigen Prozessoren wären von dieser Software ungenutzt.

Ein weiterer Grund für die Anforderung von Parallelität ist die Verwendung von Programmiersprachen, mit denen Programmabläufe einfach parallelisiert werden können.

Um die Anforderung an Parallelität zu erfüllen, muss die Software neu gestaltet werden. Dazu kommen Threads zum Einsatz, mit denen sich Programmabläufe parallelisieren lassen. Dieses Verfahren heißt Multithreading.

Bei Einsatz von Multithreading wird allerdings Threadsicherheit benötigt. Threadsicherheit beschreibt das fehlerfreie Ausführen paralleler Programmabläufe, ohne dass unvorhersehbares Verhalten, Laufzeitfehler oder Abstürze verursacht werden.

Bei der Entwicklung neuer Software greift man häufig auf System- und Anwendungsbibliotheken zurück, um vorhandene Funktionalität wiederzuverwenden. Viele Bibliotheken erfüllen die Bedingung der Threadsicherheit nicht.

Die bei SAP im SAPInst-Tool verwendete Syslib ist eine solche, nicht threadsichere, Anwendungsbibliothek. Bisher hatte die Syslib nur eine Schnittstelle zu Javascript. Da in Javascript kein Multithreading möglich ist, wurde bei Design und Implementierung der Syslib bewusst auf Threadsicherheit und Multithreading verzichtet. Aufgrund neuer Anforderungen an das SAPInst-Tool wurde die Syslib um eine Schnittstelle zu Java erweitert. Da Java Multithreading unterstützt und dies einfach eingesetzt werden kann, besteht nun die Notwendigkeit einer threadsicheren Syslib.

In dieser Diplomarbeit wird untersucht, welche Möglichkeiten es gibt, eine nicht threadsichere Anwendungsbibliothek threadsicher zu gestalten.

0.2. Hintergrund

SAP ist führender Anbieter von Unternehmenssoftware und drittgrößter Software-Produzent weltweit. SAP bietet Software für Finanzdienstleister, Öffentliche Verwaltung, Dienstleistungsbranchen, Fertigungsindustrie und Prozessindustrie an.

Angesiedelt ist die Diplomarbeit in der Software-Logistik-Abteilung von SAP. Dort wird das SAPInst-Tool entwickelt, mit dem SAP Server-Software installiert wird. Diese kann mit verschiedenen Konfigurationen auf verschiedenen Plattformen installiert werden.

Komponenten des SAPInst-Tool

Das SAPInst-Tool besteht aus folgenden Komponenten (vgl. [SAP]):

Controller

Der Controller ist die Hauptkomponente des SAPInst-Tool. Er führt sogenannte Installationskomponenten aus. Diese enthalten Installationsanweisungen in Javascript, welche in XML eingebettet sind.

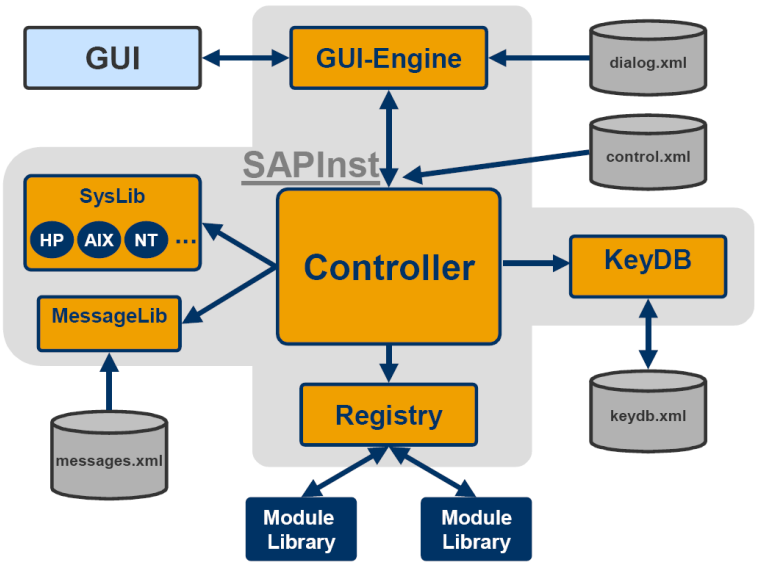


Abbildung 0.1.: SAPInst-Tool, (vgl. [SAP])

KeyDB

Die KeyDB speichert Installationsparameter in einer Datenbank ab.

GUIEngine

Die GUIEngine ist die Schnittstelle zur graphischen Oberfläche der Installation. Die Kommunikation ist über TCP/IP[1] realisiert. Dies ermöglicht eine lokale und eine ferngesteuerte Installation über die grafische Benutzeroberfläche.

MessageLib

Mit der MessageLib können Log- und Trace-Nachrichten in eine Logdatei geschrieben werden. Log-Nachrichten wie Info, Warning und Error dienen als Statusinformationen. Mit Trace-Nachrichten wird der Programmablauf aufgezeichnet.

[1] TCP/IP - Transmission Control Protocol/Internet Protocol, ein Transportprotokoll, das Datentransfer über Netzwerke ermöglicht

Syslib

Die Syslib ist eine plattformübergreifende Anwendungsbibliothek, die verschiedene Systemfunktionalitäten anbietet. Sie ist Hauptgegenstand der Untersuchungen in dieser Arbeit.

Registry

Die Registry wird verwendet, um dynamische Bibliotheken während der Laufzeit der Installation zu laden. Dies ermöglicht das Ausführen installationsspezifischer Dienste.

0.3. Ziele

Hauptaufgabe der Diplomarbeit ist die threadsichere Gestaltung der Syslib, um sie in Multithread-Umgebungen einsetzen zu können. Dazu müssen einige Umstrukturierungen in der Syslib vorgenommen werden. Nachdem der aktuelle Aufbau der Syslib untersucht wurde, werden verschiedene Methoden zur Herstellung von Threadsicherheit vorgestellt und bewertet. Dadurch kann festgestellt werden, welche Methoden sich mit dem geringsten Aufwand in die aktuelle Struktur der Syslib integrieren lassen. Diese Umstrukturierung soll teilweise ausgeführt werden, um eine Leistungsanalyse durchführen zu können.

Die Leistungsanalyse soll Änderungen in der Ausführungszeit analysieren. Erwartet wird eine erhöhte Ausführungszeit bei der Ausführung in einer Singlethread-Umgebung und eine wesentliche Verringerung der Ausführungszeit in einer Multithread-Umgebung.

Für die Durchführung der Leistungsanalyse soll eine Testumgebung erstellt werden, mit der Zeitmessungen durchgeführt werden können und die für Multithreading geeignet ist. Diese soll auch für zukünftige Multithread-Tests verwendet werden.

1. Grundlagen

In diesem Abschnitt werden die Grundlagen von Multithreading erklärt. Es wird auf Vorteile und Probleme hingewiesen, die durch Multithreading entstehen können. Ein weiteres Thema ist die Threadsicherheit. Sie wird definiert und es wird erläutert, wie man sie erreichen kann. Weiterhin gibt es eine kurze Einführung in das Gebiet der Leistungsanalyse.

1.1. Multithreading

Multithreading beschreibt das parallele Ausführen von mehreren Threads innerhalb eines Prozesses. Die Begriffe Prozess und Thread werden im folgenden Abschnitt erklärt.

1.1.1. Prozess und Thread

Prozess

Bei der Ausführung eines Programmes wird vom Betriebssystem ein Prozess erzeugt, der die Anweisungsfolgen des Programmes bearbeitet. Dabei besitzt ein Prozess einen Kontext, der Informationen über ihn enthält. Dazu gehört u.a. der Adressraum (Prozess-Speicher), der Prozess-Identifikator (PID) und die Ausführungspriorität. Bearbeitet wird ein Prozess, wenn er vom Betriebssystem Prozessorzeit zugeteilt bekommt. Dabei wird der gesamte Kontext des vorherigen Prozesses gesichert und der Kontext des neuen Prozesses geladen. Dieser Vorgang wird als Prozesswechsel oder Kontextwechsel bezeichnet.

Aufgrund des umfangreichen Kontextes werden Prozesse auch als schwergewichtig bezeichnet. (vgl. [Sch00], S. 235-240)

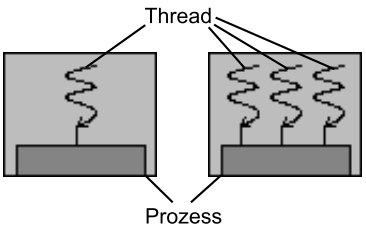


Abbildung 1.1.: Prozess und Thread

Thread

Um den Aufwand für den Kontextwechsel zu reduzieren, gibt es leichtgewichtige Prozesse, auch Threads genannt. Ein Thread ist ein Ausführungsfaden, der in einer Ausführungsumgebung (Prozess) abläuft. Somit besitzt jeder Prozess einen initialen Thread. Es können weitere Threads erzeugt werden, wobei sich alle den Prozess-Speicher teilen. Der Thread selbst besitzt einen Stack, ein Registersatz und eine Ausführungspriorität. Außerdem kann er einen lokalen Speicherbereich anfordern. Der geringe Kontext reduziert den Aufwand des Kontextwechsels erheblich, falls dieser innerhalb des Prozesses stattfindet.

Durch den gemeinsam genutzten Prozess-Speicher können die Threads miteinander kommunizieren. Eine direkte Kommunikation zwischen Threads verschiedener Prozesse ist nicht möglich (vgl. [Sch00], S. 240-241).

Die Verwendung von mehreren Threads, um Programmabläufe zu parallelisieren, wird als Multithreading bezeichnet.

1.1.2. Eigenschaften von Multithreading

Multiprozessorsysteme können Multithreading besonders gut umsetzen. So kann ein System mit zwei Prozessoren zwei Threads parallel ausführen, falls diese unabhängig voneinander sind.

Auf Systemen mit einem Prozessor kann Multithreading ebenfalls Vorteile bringen. So muss bei blockierenden Eingabe/Ausgabe-Operationen, in einem sequenziellen Programm, der gesamte Prozess angehalten werden. In einem Multithread-Programm können diese Operationen von einem Thread ausgeführt werden, während andere Threads weiterarbeiten können.

Auch in graphischen Oberflächen wird Multithreading eingesetzt. So kann auf Benutzereingaben besser reagiert werden, indem diese in separaten Threads bearbeitet werden. Moderne graphische Oberflächen, wie Java AWT[1] und Swing[2], arbeiten nach diesem Prinzip.

1.1.3. Kommunikation zwischen Threads

Kritischer Abschnitt

Die Kommunikation zwischen Threads eines Prozesses erfolgt über den Prozess-Speicher. Dabei können Daten zwischen den Threads eines Prozesses ausgetauscht werden. Dazu greifen die Threads in ihren jeweiligen Programmabschnitt auf einen gemeinsamen Speicher-Bereich zu. Dies wird als kritischer Abschnitt bezeichnet.

Synchronisation

Um gleichzeitige Zugriffe auf einen kritischen Abschnitt auszuschließen, müssen die Zugriffe synchronisiert werden. Synchronisation beschreibt die Manipulation paralleler Programmabläufe, um diese in eine bestimmte Reihenfolge zu bringen (vgl. [Sch00], S. 241). Dazu werden Synchronisationsmittel eingesetzt.

Folgende Synchronisationsmittel werden in dieser Diplomarbeit verwendet:

[1] AWT - Abstract Window Toolkit, benutzt native Grafikfunktionalität des Betriebssystemes, ist plattformunabhängig, Aussehen ist plattformabhängig

[2] Swing - Grafikfunktionalität bereitgestellt von Sun, Aussehen und Benutzung ist plattformunabhängig

Mutex

Mutex ist eine Kurzform für Mutual Exclusion (Wechselseitiger Ausschluss). Dieses Synchronisationsmittel kennt nur die zwei Zustände, gesperrt und freigegeben. Wenn dieser Mutex von einem Thread gesperrt wird, müssen alle anderen Threads auf die Freigabe warten. Auch der Thread, der den Mutex gesperrt hat, muss bei einer erneuten Anfrage auch auf die Freigabe warten. Somit würde sich dieser Thread selbst blockieren, da nur er den Mutex freigeben kann.

Rekursiver Mutex

Ein rekursiver Mutex funktioniert im wesentlichen wie ein Mutex. Der Thread, der den rekursiven Mutex sperrt, kann ihn aber auch mehrfach sperren, ohne sich selbst zu blockieren. Damit der rekursive Mutex wieder freigegeben wird, muss die Anzahl der Freigaben gleich der Anzahl der Sperraufrufe sein.

Semaphore

Es gibt zwei Arten von Semaphoren, den binären Semaphore und den Zähl-Semaphore. Der binäre Semaphore hat, ähnlich wie der Mutex, zwei Zustände. Er kann einmal gesperrt und freigegeben werden. Der Zähl-Semaphore gibt eine bestimmte Anzahl vor, wie oft er gesperrt werden kann. Ist diese Anzahl erreicht, müssen alle anderen Threads auf die Freigabe des Semaphores warten. Jeder Thread kann einen Semaphore freigeben, unabhängig davon welcher Thread den Semaphore gesperrt hat.

Lese/Schreib Sperre

Die Lese/Schreib-Sperre kennt zwei gesperrte Zustände. Eine Lese-Sperre kann beliebig oft von beliebig vielen Threads gesperrt werden, ohne andere lesende Threads zu blockieren. Eine Schreib-Sperre kann nur von einem Thread aktiviert werden, wenn keine Lese- und keine Schreib-Sperre aktiv ist. Ist die Schreib-Sperre einmal aktiv, müssen alle Threads, die eine Lese- oder Schreib-Sperre anfordern, auf die Freigabe der Schreib-Sperre warten.

Event

Ein Event ermöglicht die Kommunikation und Synchronisation von Threads untereinander. Dazu kann ein Thread ein Event auslösen und somit anderen, auf dieses Event wartenden, Threads signalisieren, dass sie weiterarbeiten können. Dadurch kann nicht nur ein kritischer Abschnitt geschützt, sondern auch der Programmablauf gesteuert werden.

1.2. Threadsicherheit

1.2.1. Definition

Threadsicherheit beschreibt das korrekte Verhalten einer Klasse, einer Methode oder eines Code-Bereiches, auf den mehrere Threads parallel zugreifen können, ohne dass dabei zusätzlicher Synchronisationsaufwand benötigt wird. Dem zugrunde liegt der Begriff der Korrektheit. Ein korrektes Verhalten wird als das Verhalten definiert, welches die Erwartungen an eine vorgegebene Spezifikation erfüllt.

Dies bedeutet, dass der Zugriff durch mehrere Threads auf einen abrufbaren und veränderbaren Status geschützt werden muss. Beispiele für einen solchen Status sind globale Variablen oder Attribute einer Klasse. Eine threadsichere Klasse beinhaltet alle benötigten Synchronisationsmittel, um den Zugriff auf den Status der Klasse zu schützen, ohne dass der Benutzer eigene Synchronisationsmittel benötigt. Aus der Definition folgt auch, dass statusfreie Klassen grundsätzlich threadsicher sind (vgl. [Goe07], S. 17-19).

1.2.2. Race Condition

Da der Zugriff auf einen Status durch mehrere Threads erfolgen kann, ist dies ein kritischer Abschnitt. Falls dieser Zugriff nicht synchronisiert wird, können Race Conditions auftreten.

Bei einer Race Condition greifen zwei oder mehr Threads gleichzeitig auf einen kritischen Abschnitt zu, wobei das Ergebnis bei jedem Durchlauf unterschiedlich ausfallen kann. Zum Beispiel kann die Berechnung einer Variablen zu verschiedenen Ergebnissen führen.

Folgende Berechnungen sollen durchgeführt werden: gegeben ist $X = 10$. X ist eine Variable, die zwischen zwei Threads geteilt wird. Es sollen die Berechnungen $X + 10$ und $X - 5$ durchgeführt werden. In einem sequentiellen Programm lautet das Ergebnis $X = 15$. Teilt man diese Berechnungen den zwei Threads zu, können unterschiedliche Ergebnisse entstehen.

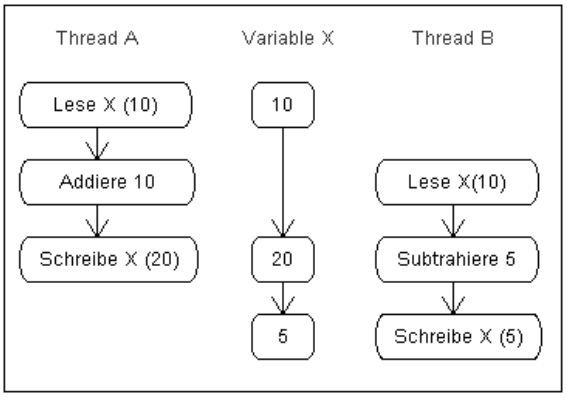


Abbildung 1.2.: Thread A liest zuerst

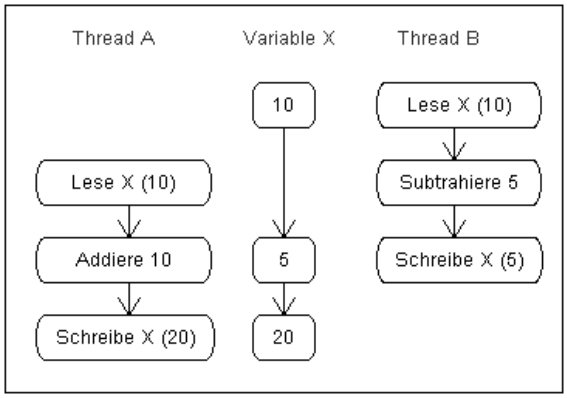


Abbildung 1.3.: Thread B liest zuerst

Atomare Operation

Der Grund, für die unterschiedlichen Ergebnisse im oberen Beispiel liegt darin, dass die Berechnungen nicht als atomare Operationen ausgeführt werden. Eine atomare Operation ist die Zustandsabfrage oder -änderung eines Status, die nicht teilbar ist. Nur ein Thread darf den Zustand des Status abfragen oder verändern. Alle anderen Threads müssen auf den Abschluss der Operation warten.

Um das oben beschriebene Problem zu lösen, muss der kritische Abschnitt zu einer atomaren Operation werden. Dabei kommen Synchronisationsmittel zum Einsatz. Im Beispiel wird somit der Zugriff auf die Variable X eingeschränkt. Es darf nur ein Thread die Variable X lesen und verändern.

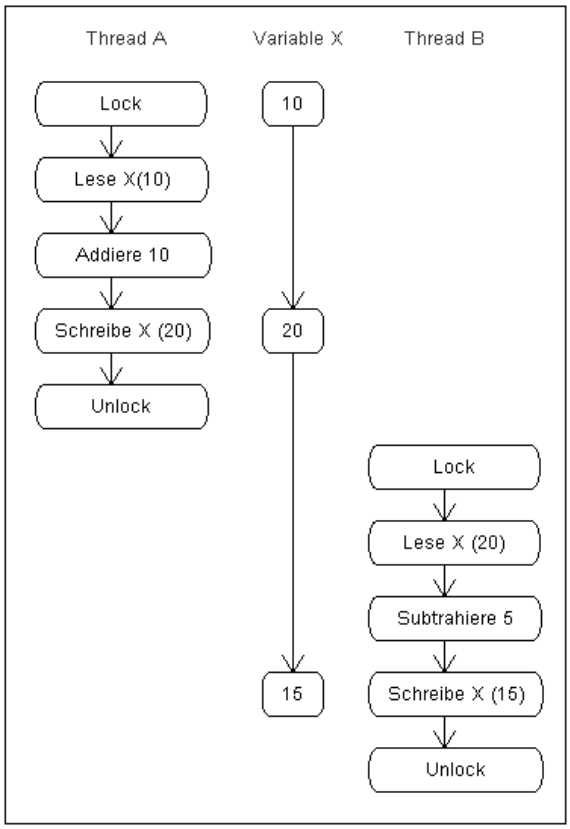


Abbildung 1.4.: Geschützter Zugriff

Durch Synchronisation mit *Lock* und *Unlock* wird der sich dazwischen befindliche kritische Abschnitt zu einer atomaren Operation, die den Zugriff auf die Variable X synchronisiert. Das Ergebnis beträgt nun $X = 15$, egal ob Thread A oder Thread B den kritischen Abschnitt zuerst betritt. (vgl. [Car06], S. 25-26).

1.2.3. Deadlock - Verklemmung

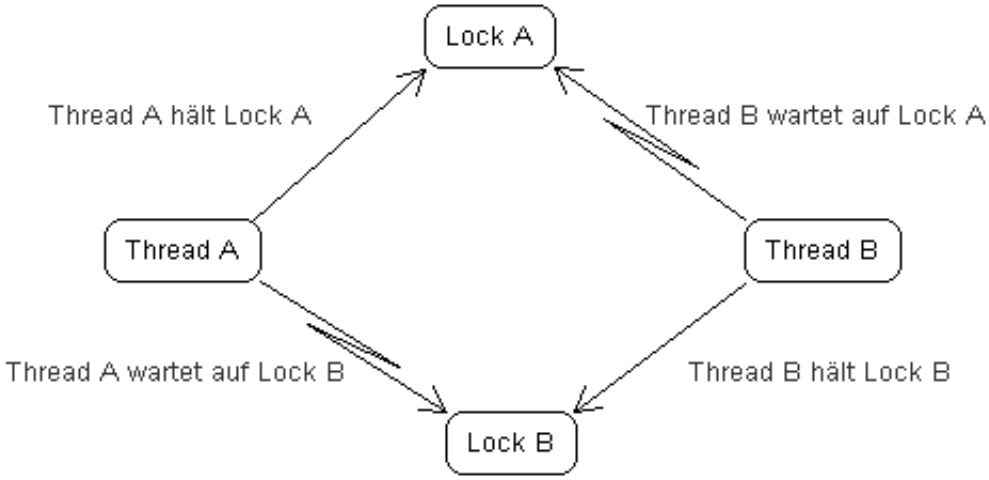


Abbildung 1.5.: Verklemmung

Eine Verklemmung entsteht, wenn zwei oder mehr Threads sich gegenseitig blockieren. Die häufigste Ursache für eine Verklemmung sind mehrere Threads, die jeweils ein Synchronisationsmittel halten und ein anderes benötigen, das von einem anderen Thread gehalten wird.

Behandlung von Verklemmung

Folgende Bedingungen müssen erfüllt sein, damit eine Verklemmung entsteht.

1. Wechselseitiger Ausschluss: Die Threads sperren sich gegenseitig, um exklusiv auf einen kritischen Abschnitt zuzugreifen
2. Halte- und Wartebedingung: Threads sperren mehrere Synchronisationsmittel, ohne die bereits gesperrten freizugeben
3. Nicht Unterbrechbar: Ein gesperrtes Synchronisationsmittel kann nur von dem sperrenden Thread freigegeben werden
4. Zyklische Wartebedingung: Die Threads sperren verschiedene Synchronisationsmittel in unterschiedlicher Reihenfolge, so das eine geschlossene Kette entsteht (siehe Abbildung 1.5)

Dabei sind die ersten drei Bedingungen notwendig, das heißt auch wenn diese Bedingungen erfüllt sind, muss es nicht zu einer Verklemmung kommen. Die vierte Bedingung ist hinreichend. Falls diese eintritt, ist eine Verklemmung entstanden.

Es gibt drei Ansätze, um eine Verklemmung zu lösen:

- Verhinderung: Einen Verklemmung kann verhindert werden, indem eine Bedingung ausgeschlossen wird.
- Vermeidung: Das Betriebssystem prüft die Sperranforderung eines Synchronisationsmittels und entscheidet, ob es gesperrt werden darf oder nicht.
- Entdeckung und Beseitigung: Dabei wird eine Verklemmung zugelassen, vom Betriebssystem erkannt und verschiedene Maßnahmen durchführt, um die Verklemmung zu lösen.

(vgl. [Sch00], S. 250).

1.3. Leistungsanalyse

Die Leistungsanalyse befasst sich mit der Analyse und Beurteilung des Leistungsverhaltens von Programmen. Dazu werden verschiedene Methoden eingesetzt. Näher betrachtet werden Benchmarking und Leistungsmessung (vgl. [Han95], S. 19).

1.3.1. Grundbegriffe

Als Grundbegriffe der Leistungsanalyse werden die Begriffe Leistung, Speedup und Effizienz eingeführt.

Leistung und Leistungsindex

Die Leistung kann als Wert eines Systems für einen Benutzer verstanden werden. Ein System umfasst Hardware und Software, die benötigt wird, um eine Aufgabe zu erledigen.

Der Leistungsindex ist eine messbare Eigenschaft eines Systems. Die Prozessorgeschwindigkeit oder die Leistungsfähigkeit des Arbeitsspeichers sind zwei Beispiele für Leistungsindizes von Hardware. Eine Wichtung gibt an, wie stark der jeweilige Leistungsindex in die Gesamtleistung einfließt.

Berechnen lässt sich die Leistung aus der Summe der Produkte eines Leistungsindex mit einer Wichtung.

Leistung nach Kesselmann:

$$P = \sum p_i * w_i$$

p_i ... Leistungsindex
w_i ... Wichtung

Speedup

Ein Leistungsindex für ein Programm ist der Speedup. Dieser gibt den Grad der Parallelisierung, in Abhängigkeit von der Anzahl N der Prozessoren, des Programmes an. Im Idealfall ist der Speedup gleich N.

$$S(N) = \frac{t_S}{t_N}$$

N ... Anzahl Prozessoren

t_S ... Ausführungszeit einer sequenziellen Version des Programmes

t_N ... Ausführungszeit einer parallelen Version des Programmes

Effizienz

Aus dem Speedup lässt sich die Effizienz berechnen. Dazu wird der Speedup mit der Anzahl der Prozessoren normiert.

$$E = \frac{S(N)}{N}$$

$S(N)$ ... Speedup

N ... Anzahl der Prozessoren

1.3.2. Methoden der Leistungsanalyse

Benchmarking

Benchmarking ist eine Methode der Leistungsanalyse, mit der Rechnersysteme verglichen werden können. Dabei wird ein Referenzprogramm oder ein Satz von Referenzprogrammen eingesetzt. Bei diesen handelt es sich um Test-Programme, die ein Rechnersystem nach verschiedenen Kriterien bewerten. Die Ergebnisse werden in Relation gesetzt, um somit das Leistungsverhalten der Kombinationen von unterschiedlicher Hardware und Software zu vergleichen (vgl. [Han95], S. 24).

Leistungsmessung

Die Leistungsmessung ist die Beobachtung des Ablaufverhaltens eines Programmes. Die gewonnenen Erkenntnisse aus diesen Beobachtungen werden häufig zur Optimierung des Programmes genutzt. Die Daten können auf drei verschiedenen Ebenen des Systems gemessen werden:

- Hardwareebene: Übertragungsraten der Kommunikationsmedien, Anzahl der Treffer bei Zugriffen auf Pufferspeicher, Auslastung der Prozessoren
- Betriebssystemebene: Übertragungsraten von logischen Kommunikationsverbindungen, Prozessornutzung durch einen Prozess oder Thread
- Programmebene: Aufrufhäufigkeit von Funktionen, Zeilen oder Schleifen, Prozessornutzung einer Funktion

Die Daten können mit Leistungsmesssystemen erfasst werden. Dabei gibt es zwei Arten von Leistungsmesssystemen. Bei Online-Systemen erfolgt die Datenerfassung gleichzeitig mit der Datengenerierung und Datendarstellung. Offline-Systeme erfassen die Daten und werten diese nach der Messung aus (vgl. [Han95], S. 28 - 30).

2. Problemanalyse

In diesem Kapitel wird näher auf den Aufbau der Syslib eingegangen. Es werden die Komponenten und Mechanismen der Syslib vorgestellt und anschließend auf Threadsicherheit untersucht.

2.1. Syslib

Die Syslib ist eine C++-Anwendungsbibliothek. Sie kapselt Betriebssystemfunktionalität in einer objektorientierten Klassenstruktur. Folgende Plattformen werden von der Syslib unterstützt:

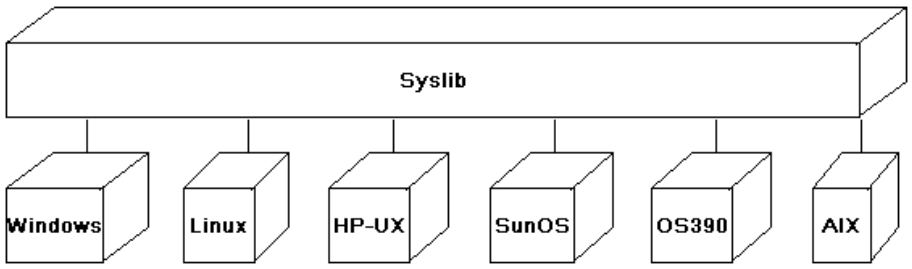


Abbildung 2.1.: Von der Syslib unterstützte Betriebsysteme

Es werden nicht alle Funktionen auf allen Plattformen angeboten. Zum Beispiel können Dateisystem-Links nur auf Unix-Systemen verwaltet werden und die Verwendung der Registrierung ist nur auf Windows möglich.

2.1.1. Komponenten der Syslib

Die Syslib besteht aus folgenden Komponenten:

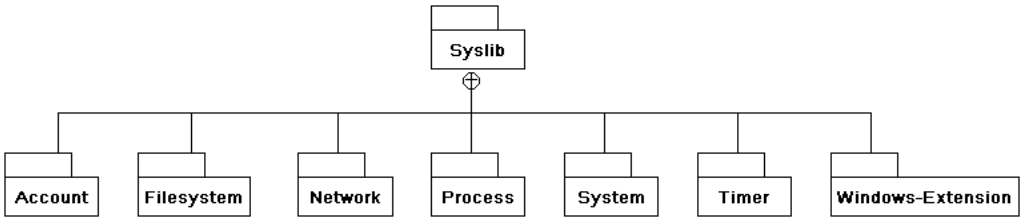


Abbildung 2.2.: Komponenten der Syslib

Account - Benutzer- und Gruppenkonten

Es können Benutzer und Gruppen erstellt, gelöscht und bearbeitet werden. Weiterhin ist eine Rechteverwaltung dieser Konten möglich.

Filesystem - Dateisystem

Diese Komponente ermöglicht die Verwaltung des Dateisystems. Sie kann Dateien und Verzeichnisse erstellen, löschen, verschieben, kopieren und Rechte setzen. Des Weiteren können Ordnerfreigaben verwaltet und andere Dateisysteme über Netzwerk eingehängt werden.

Network - Netzwerk

Mit der Netzwerk-Komponente können Informationen wie IP-Adressen[1], Rechnernamen und Domänennamen abgefragt werden. Des Weiteren können Netzwerkdienste hinzugefügt werden.

Process - Prozessverwaltung

Die Prozess-Komponente ermöglicht es, externe Programme auszuführen und deren Status abzufragen. Es können auch die aktuellen Prozess-Umgebungsvariablen abgefragt werden.

[1] IP - Internet Protokoll: Grundlage für die Adressierung von Computern in Netzwerken

System - Systeminformationen

Hiermit können allgemeine Systeminformationen wie die Größe des Hauptspeichers oder der freie Festplattenspeicher bestimmt werden.

Timer - Zeitmessung

Mit dieser Komponente können einfache Zeitmessungen durchgeführt werden.

Windows-Extensions - Registrierung und Dienste von Windows

Diese Komponente erlaubt das Erstellen, Bearbeiten und Löschen von Registrierungsschlüsseln und -werten sowie deren Rechteverwaltung. Weiterhin können Windows Dienste erstellt, bearbeitet, gelöscht, gestartet und beendet werden.

2.1.2. Aufbau der Syslib

In der Syslib werden verschiedene Entwurfmustern verwendet.

Entwurfsmuster Brücke

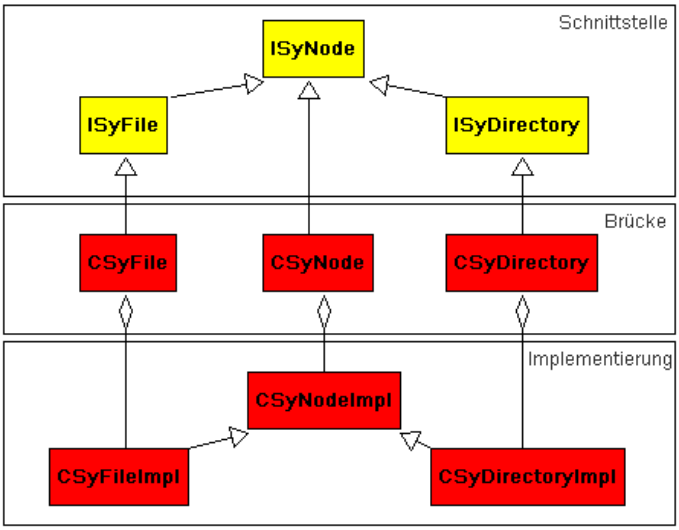


Abbildung 2.3.: Beispiel aus der Dateisystem-Komponente

Die Klassenstruktur der Syslib ist nach dem Entwurfsmuster der Brücke aufgebaut. Diese besteht aus einer Schnittstellenschicht, einer Brückenschicht und einer Implementierungsschicht. Die Schnittstellen bieten dem Benutzer die Funktionalität der Syslib an. Die Brückenklassen sind Ableitungen der Schnittstellen und enthalten eine Instanz ihrer jeweiligen Implementierungsklasse. Diese enthalten die eigentliche Implementierung der Funktionalität. Somit kapseln die Brückenklassen die Implementierung von ihrer Schnittstelle. Dies ermöglicht eine Änderung der Implementierung, ohne dass der Benutzer der Schnittstelle durch diese Änderung betroffen ist.

Die Instanzen der Brückenklassen werden als Syslib-Objekte bezeichnet.

Referenzzählung und intelligente Zeiger

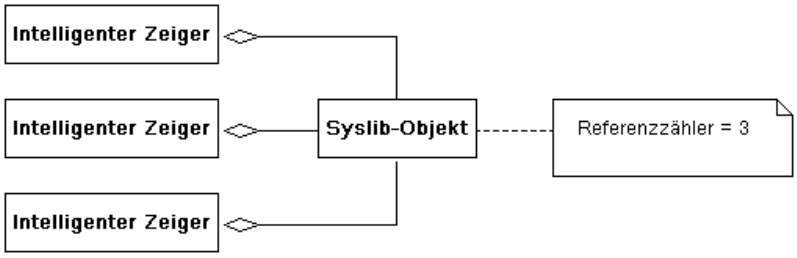


Abbildung 2.4.: Intelligenter Zeiger

Referenzzählung ist ein interner Mechanismus zur automatischen Speicherverwaltung von erzeugten Syslib-Objekten. Dabei enthält ein Syslib-Objekt einen Referenzzähler, der die Anzahl der aktuellen Referenzen auf das Objekt angibt.

Die Automatisierung der Referenzzählung erfolgt durch einen intelligenten Zeiger. Dies ist eine Klasse, die die Funktion eines Zeigers nachahmt und als Attribut einen Zeiger auf ein Syslib-Objekt besitzt. Beim Erstellen eines intelligenten Zeiger-Objektes wird der Referenzzähler, des vom Zeiger referenzierten Objektes, um eins erhöht. Wenn der intelligente Zeiger gelöscht wird, verringert er den Referenzzähler um eins. Falls dieser Null erreicht, existiert keine Referenz mehr auf das Syslib-Objekt und es wird gelöscht.

Fabrikklasse

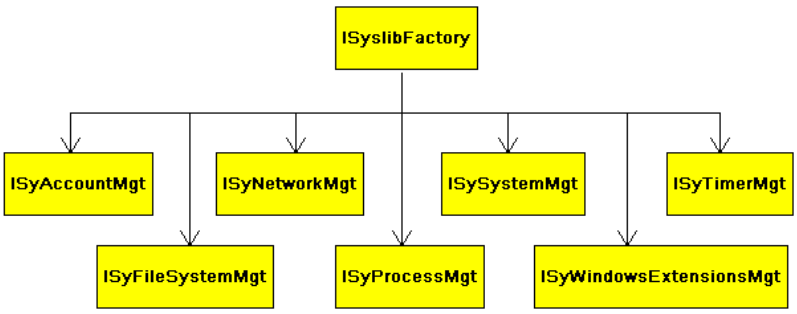


Abbildung 2.5.: Fabrikklassen der Syslib

Die Syslib verwendet Fabrikklassen, um Syslib-Objekte zu erstellen. Diese sind, wie die anderen Syslib-Klassen, mit einer Brücken-Struktur aufgebaut. Es gibt eine allgemeine Syslib-Fabrik, die Fabriken für die verschiedenen Syslib-Komponenten erstellen kann. Die Rückgabewerte der Funktionen der Fabrikklassen sind intelligente Zeiger auf die Schnittstelle des angeforderten Syslib-Objektes. Die Fabrikklassen bieten neben der Erstellung von Syslib-Objekten auch komponentenspezifische Funktionalität an.

Singleton

Die Fabrikklassen der Syslib sind als Singleton implementiert. Ein Singleton ist eine Klasse, von der nur eine Instanz angelegt wird. Alle Verwendungen dieses Singletons erfolgen über diese Instanz.

Iterator

Als Rückgabewert auf eine Datenkollektion werden Iteratoren verwendet. Nach der Erstellung eines Iterators referenziert dieser auf den ersten Eintrag, der nun ausgelesen und bearbeitet werden kann. Durch Weiterschaltung zeigt der Iterator auf den nächsten Eintrag. Zudem kann geprüft werden, ob noch ein weiterer Eintrag existiert. Somit wird ein schrittweises Zugreifen auf alle Daten einer Kollektion erlaubt, unabhängig von der internen Darstellung der Daten.

2.2. JSyslib - Java Schnittstelle

Die JSyslib ist die Java-Schnittstelle für die Syslib. Sie implementiert keine eigene Funktionalität, sondern leitet die Funktionen an die Syslib weiter. Dabei ist sie, wie die Syslib, in einer Brücken-Struktur aufgebaut. Der Benutzer verwendet die Schnittstellen der JSyslib. Die Brückenklassen, die von den Schnittstellen abgeleitet sind, leiten die Funktionsaufrufe an die Syslib weiter. Diese repräsentiert die Implementierungsschicht.

Anbindung von Java an C++ (JSyslib/Syslib)

Realisiert ist die Anbindung der JSyslib an die Syslib über SWIG[2]. Dabei handelt es sich um einen Schnittstellen-Compiler, der C/C++-Funktionalität in anderen Programmiersprachen bereitstellen kann. Dazu wird der C/C++-Quelltext nach verschiedenen Mustern analysiert und daraus eine Verschalung für die jeweilige Sprache generiert. (vgl. [SWIG])

Um die Syslib-Funktionen in der JSyslib bereitzustellen, werden die C++-Klassen der Syslib auf Java-Klassen abgebildet:

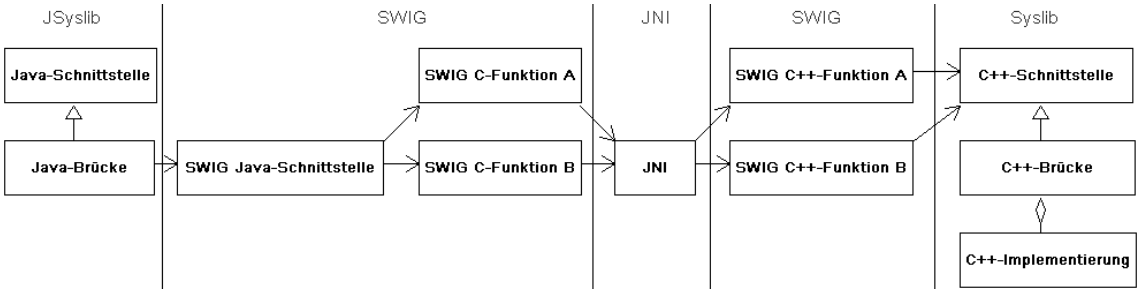


Abbildung 2.6.: Funktionsweise von SWIG

Der Aufruf einer Syslib-Funktion, durch eine Java-Anwendung, erfolgt über die SWIG Java-Schnittstelle. Dabei handelt es sich um eine native Java-Funktion. Der Funktionsaufruf läuft nun über eine C-Funktion, deren Name sich aus dem Klassennamen, dem Funktionsnamen und den Parametertypen der Syslib-Funktion zusammensetzt. Die C-Funktion

[2]SWIG - Simplified Wrapper and Interface Generator, Vereinfachter Verschalungs- und Schnittstellen-Generator

leitet den Aufruf an die JNI[3]-Schicht weiter. Diese übersetzt die Java-Datentypen in C-Datentypen. In den SWIG C++-Funktionen werden die Datentypen für C++ ausgelesen und an die Klassen der Syslib weitergegeben.

Beim Beenden der Funktion, wird ein eventueller Rückgabewert im umgekehrter Ablaufreihenfolge nach Java zurück konvertiert.

[3]JNI - Java Native Interface, eine Bibliothek zur Konvertierung von Java- in C-Funktionsaufrufe und umgekehrt

2.3. Untersuchung auf Threadsicherheit

Folgende Programmteile der Syslib und der von ihr verwendeten Komponenten müssen auf Threadsicherheit untersucht werden:

MessageLib

Die MessageLib ist eine externe Komponente, die von der Syslib zum Schreiben von Logdateien verwendet wird. In einer Multithread-Umgebung müssen zwei Dinge beachtet werden. Zum einen muss die MessageLib selbst threadsicher sein, zum anderen würde es die Lesbarkeit der Logdatei verringern, wenn mehrere Threads in unterschiedlicher Reihenfolge die Nachrichten schreiben. Letzteres ist zwar kein kritisches Multithread-Problem, würde aber eine Fehlersuche durch die Analyse der Logdatei erschweren.

Eine genauere Untersuchung und Bearbeitung der MessageLib wurde nicht im Rahmen der Diplomarbeit durchgeführt. Allerdings wurde auf Basis dieser Arbeit eine threadsichere Version der MessageLib erstellt.

Referenzzählung

Die Referenzzählung ist nicht threadsicher, da der Referenzzähler nicht geschützt wird. Bei der Verwendung von mehreren Threads kann es passieren, dass diese gleichzeitig einen intelligenten Zeiger löschen und somit den Referenzzähler um eins verringern. Dabei werden Lese- und Schreiboperationen von mehreren Threads gleichzeitig ausgeführt. Im ungünstigen Fall kommt es zu einer Race Condition, wobei zwei Threads den gleichen Wert lesen, diesen um eins verringern und das Ergebnis zurückschreiben. Dadurch geht eine Verringerung verloren und das Syslib-Objekt besitzt fälschlicherweise noch eine Referenz. Dies führt dazu, dass das Syslib-Objekt nie gelöscht wird und ein Speicherleck entsteht.

Singleton

Singletons in Multithread-Umgebungen sind threadsicher, wenn sie keinen Status besitzen oder über Synchronisationsmittel verfügen, die Zugriffe auf den Status schützen.

Bei den in der Syslib verwendeten Singletons entsteht ein anderes Problem. Die Syslib ist eine dynamische Bibliothek, die auch zur Laufzeit geladen wird. In einer Multithread-Umgebung kann sie somit mehrfach, von mehreren Threads, geladen werden. Dabei kann bei jedem Ladevorgang ein neues Singleton erzeugt werden. Da die Syslib diese auch als globale Datenspeicher verwendet, können nun mehrere solcher globaler Datenspeicher existieren. Diese Inkonsistenz führt nicht nur zu Leistungsproblemen, sondern auch zu undefiniertem Verhalten.

Iteratoren

Iteratoren können auf verschiedene Weise implementiert sein. Zum einen können die Daten beim Erstellen eines Iterators kopiert werden. Dies stellt kein Problem dar, da jeder Thread auf einer Kopie der Daten iteriert.

Eine andere Möglichkeit ist das Iterieren auf Referenzen. Ein Beispiel ist ein Iterator auf einer Verzeichnisstruktur, die direkt auf das Dateisystem referenziert. Diese Art der Iteratoren ist problematisch. Wenn mehrere Threads den selben Iterator benutzen, kann jeder Thread den Iterator weiterschalten. Zum einen folgt daraus eine unvollständige Iteration, da andere Threads den Iterator bereits weitergeschaltet haben können, während die angeblich nächsten Daten gelesen werden. Zum anderen ist das Verhalten eines Threads nicht vorhersehbar, wenn ein anderer Thread die referenzierten Daten verändert oder löscht. Das Verhalten ist ebenfalls unvorhersehbar, wenn ein Thread auf den vom Iterator referenzierten Daten arbeitet, während ein anderer Thread diesen weiter schaltet.

Systemfunktionen

Bei Systemfunktionen gibt es threadsichere und nicht threadsichere Varianten. Das Verwenden von nicht threadsicheren Systemfunktionen in einer Multithread-Umgebung kann ebenfalls zu undefiniertem Verhalten führen. Häufig ist auch nicht eindeutig erkennbar, ob es sich um eine threadsichere oder nicht threadsichere Systemfunktion handelt.

JSyslib

Der größte Teil der JSyslib besteht aus generiertem SWIG-Code. Dieser ist threadsicher, weil die SWIG-Klassen die Funktionen nur weiterleiten und keinen Status besitzen.

Solange ein Benutzer ein JSyslib-Objekt referenziert, existiert ebenfalls ein Syslib-Objekt. Dieses wird von der jeweiligen Brückenklasse der JSyslib verwaltet. Dies bedeutet, dass die Brückenklassen einen Status besitzen. Somit muss nicht nur der Zugriff auf diesen Status geschützt werden, sondern auch der Löschvorgang des Syslib-Objektes.

Die threadsichere Gestaltung der JSyslib wurde nicht im Rahmen der Diplomarbeit durchgeführt und wird deshalb nicht weiter untersucht.

3. Konzeption

In diesem Kapitel wird auf die Möglichkeiten, Threadsicherheit herzustellen, eingegangen. Dazu wird eine Bibliothek entworfen, die Thread- und Synchronisationsmittel bereitstellt. Danach werden Modelle zur threadsicheren Gestaltung der Syslib vorgestellt. Des Weiteren wird ein Test-Framework für Leistungstests entworfen. Es werden Tests konzipiert, die das Verhalten der Syslib vor und nach der Einführung von Synchronisationsmaßnahmen untersuchen. Außerdem soll die Multithread-Fähigkeit der Syslib untersucht werden.

3.1. ThreadLib

Um Threads und Synchronisationsmittel objektorientiert verwenden zu können, wird eine separate Klassenstruktur benötigt. Dies soll durch das Projekt ThreadLib erfolgen.
Die ThreadLib hat zwei Hauptaufgaben:

- Thread-Modul: Verwaltung von Thread-Objekten
- Synchronisationsmittel-Modul: Verwalten von Synchronisationsmitteln

3.1.1. Thread-Modul

Den Einstiegspunkt für das Thread-Modul bildet die Fabrikklasse *IThrThreadFactory* die über die globale Methode *getThrThreadFactory()* erzeugt werden kann. Im folgenden Abschnitt werden die Klassen des Thread-Moduls vorgestellt.

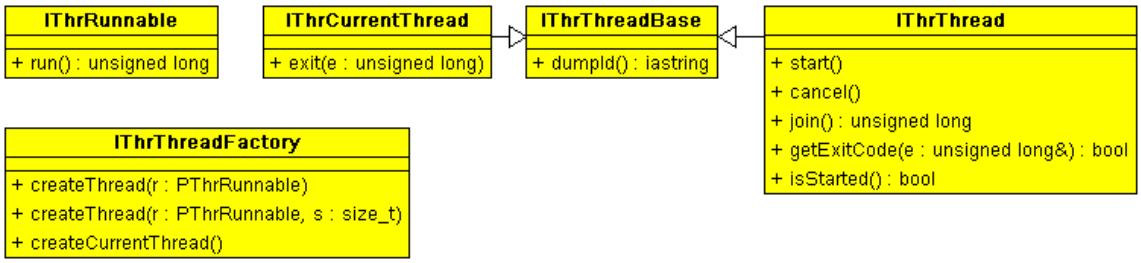


Abbildung 3.1.: Thread Modul

IThrRunnable

Die Methode *run()* aus der Schnittstellenklasse *IThrRunnable* ist der Einstiegspunkt für einen Thread. Diese muss der Benutzer selbst implementieren.

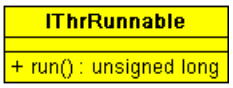


Abbildung 3.2.: *IThrRunnable*

IThrThreadFactory

Diese Fabrikklasse besitzt drei Methoden. Die Methode *createThread(PThrRunnable)* erzeugt ein Thread-Objekt mit einem selbst definierten *IThrRunnable*-Objekt. Die zweite Methode *createThread(PThrRunnable, size_t)* entspricht der ersten Methode, wobei der zusätzliche Parameter die Stack-Größe des Threads festlegt.

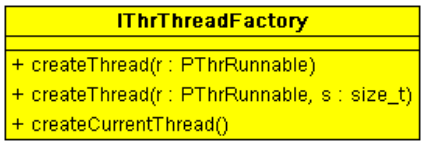


Abbildung 3.3.: *IThrThreadFactory*

Mit *createCurrentThread()* wird ein Thread-Objekt erzeugt, welches verschiedene Informationen über den aktuellen Thread zurückgeben kann.

IThrThread

Die Klasse *IThrThread* ist die Kontrollklasse für einen Thread. Die Methode *start()* erzeugt und startet den Thread mit der, in *IThrRunnable* definierten, *run()* Methode. Mit der Methode *cancel()* kann der Thread abgebrochen werden. Dies ist eine kritische Funktion, da der Status des laufenden Threads meist nicht bekannt ist. Dies kann zu einer Verklemmung führen, wenn der beendete Thread noch Synchronisationsmittel hält, diese aber nicht mehr freigeben kann.

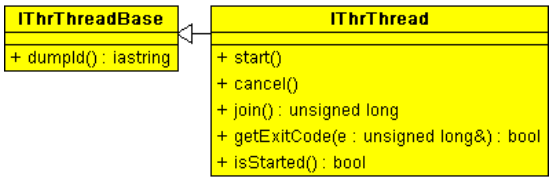


Abbildung 3.4.: *IThrThread*

Um auf das Beenden eines Thread T_r zu warten, ist *join()* zu verwenden. Damit wird der Thread T_j, der die Funktion *join()* ruft, angehalten. Wenn der Thread T_r beendet wird, gibt die Funktion *join()* den Rückgabewert von Thread T_r zurück.

Mit der Methode *getExitCode()* kann der Rückgabewert eines beendeten Threads abgerufen werden.

Die Methode *isStarted()* gibt den Wert *true* zurück, falls der Thread bereits gestartet wurde, sonst *false*.

IThrCurrentThread

Die Klasse *IThrCurrentThread* besitzt zwei Methoden, die den aktuellen Thread betreffen. Die Methode *dumpId()* gibt die interne Thread-Identifikationsnummer als Zeichenkette zurück.

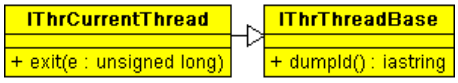


Abbildung 3.5.: *IThrCurrentThread*

Mit *exit(unsigned long)* wird der Thread beendet. Der als Parameter angegebene Wert legt dem Rückgabewert des Threads fest.

3.1.2. Synchronisationsmittel-Modul

Die Fabrikklasse *IThrSyncFactory* ist der Einstiegspunkt für das Synchronisationsmittel-Modul und kann über die globale Methode *getThrSyncFactory()* erzeugt werden. Die Fabrikklasse erstellt Mutexe, Semaphore, Lese/Schreib-Sperren, Events und Dummy-Sperren.

Im folgenden Abschnitt werden die Klassen des Synchronisationsmittel-Moduls vorgestellt.

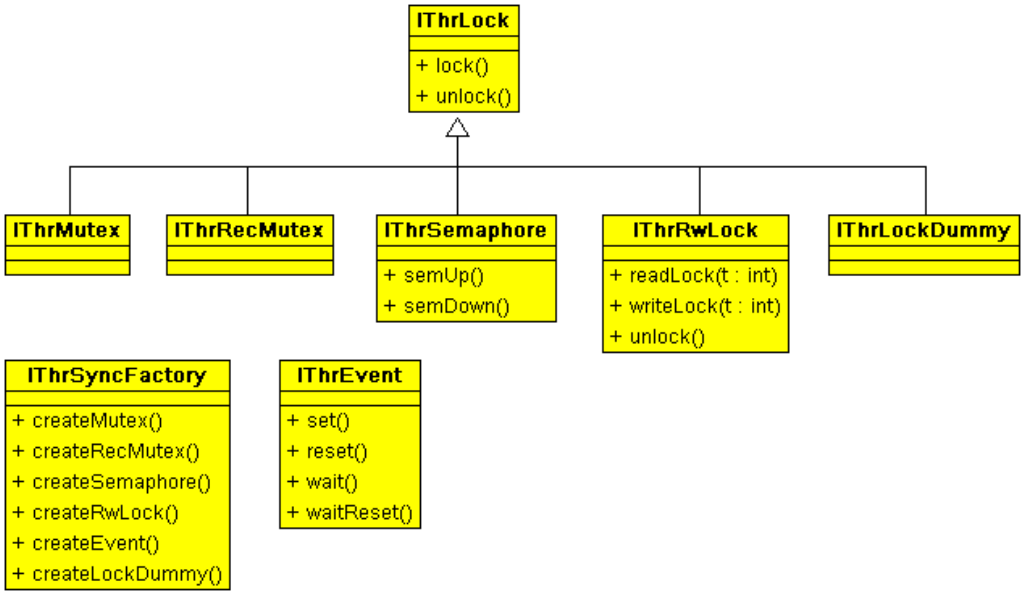


Abbildung 3.6.: Synchronisationsmittel-Modul

IThrSyncFactory

Diese Fabrikklasse erzeugt Synchronisationsmittel. Die Methode *createMutex()* erstellt einen einfachen Mutex, der die zwei Zustände gesperrt und freigegeben kennt. Mit *createRecMutex()* wird ein rekursiver Mutex erzeugt. Dieser Mutex kann mehrfach vom selben Thread gesperrt werden.

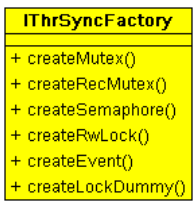


Abbildung 3.7.: *IThrSyncFactory*

Ein binärer Semaphore wird mit der Methode *createSemaphore()* erzeugt. Einen Zähl-Semaphore erhält man, wenn man dieser Funktion einen Zählwert übergibt.

Die Methode *createRwLock()* erzeugt eine Lese/Schreib-Sperre. Ein Event kann mit der Methode *createEvent()* erstellt werden.

Mit der Methode *createLockDummy()* wird ein leer implementiertes Synchronisationsmittel erzeugt.

IThrLock

Die Klasse *IThrLock* ist die Basisklasse für die Synchronisationsmittel. Sie besitzt die Methoden *lock()* und *unlock()*, die von den abgeleiteten Klassen überschrieben werden.

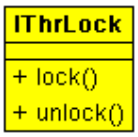


Abbildung 3.8.: *IThrLock*

IThrMutex und *IThrRecMutex*

Der einfache und der rekursive Mutex werden über die Klassen *IThrMutex* und *IThrRecMutex* durch die Methoden *lock()* und *unlock()* gesperrt und freigegeben.

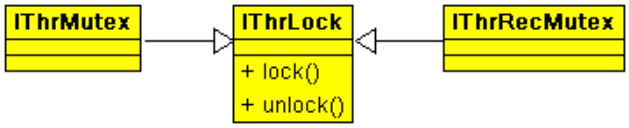


Abbildung 3.9.: *IThrMutex* und *IThrRecMutex*

IThrSemaphore

Die Klasse *IThrSemaphore* besitzt zwei weitere Methoden *semUp()* und *semDown()*. Zum Sperren kann die Methode *semDown()* oder *lock()* verwendet werden. Zum Freigeben wird *semUp()* oder *unlock()* verwendet.

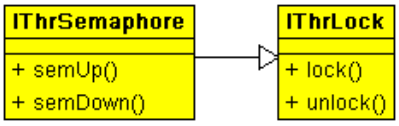


Abbildung 3.10.: *IThrSemaphore*

IThrRwLock

Die Lese/Schreib-Sperre *IThrRwLock* besitzt zusätzlich zu *lock()* und *unlock()* die Methoden *readLock()* und *writeLock()*. Die Lese-Sperre *readLock()* wird von allen Threads verwendet, die lesend auf einen kritischen Abschnitt zugreifen wollen. Die Schreib-Sperre *writeLock()* wird von einem Thread gesperrt, um auf einen kritischen Abschnitt exklusiv schreibend zugreifen zu können. Mit *unlock()* können beide Sperren freigegeben werden. Die abgeleitete Methode *lock()* entspricht der Methode *writeLock()*.

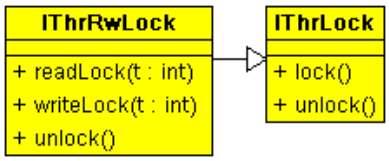


Abbildung 3.11.: *IThrRwLock*

IThrEvent

IThrEvent leitet nicht von *IThrLock* ab, da sich die Funktionalität der Events nicht auf einfaches Sperren und Freigeben abbilden lässt. Die Methode *set()* löst das Event aus. Alle Threads, die mit den Methoden *wait()* oder *waitReset()* auf dieses Event warten, reagieren nun. Falls ein Thread mit *waitReset()* wartet, wird das Event sofort zurückgesetzt, damit nur der *waitReset()* rufende Thread fortfahren kann. Beiden Methoden kann eine maximale Wartezeit in Millisekunden übergeben werden. Mit *reset()* wird das Event manuell zurückgesetzt.

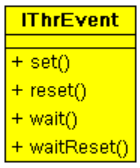


Abbildung 3.12.: *IThrEvent*

IThrLockDummy

Die Klasse *IThrLockDummy* ist eine leer implementierte Sperre. Sie kann für alle von *IThrLock* abgeleiteten Synchronisationsmittel eingesetzt werden. Anwendung findet diese Sperre in Singlethread-Umgebungen, um Synchronisationsaufwand zu vermeiden.

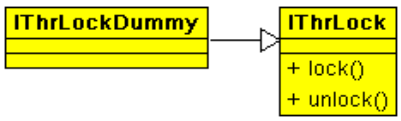


Abbildung 3.13.: *IThrLockDummy*

3.1.3. Bereichssperre

Eine weitere Funktionalität der ThreadLib ist die Bereichssperre. Damit kann ein Bereich, zum Beispiel ein kritischer Abschnitt, geschützt werden. Dazu wird eine lokale Variable vom Typ *CThrLocker* mit einem Synchronisationsmittel vom Typ *IThrLock* als Parameter angelegt. Die Sperre wird im Konstruktor der Klasse *CThrLocker* automatisch gesperrt und in deren Destruktor wieder freigegeben. Solange diese Variable existiert ist der entsprechende Bereich geschützt.

Ein Vorteil einer Bereichssperre ist die Vermeidung von Programmierfehlern, die durch Vergessen des Aufrufes einer *lock()* oder *unlock()* Methode entstehen können.

Weiterhin ist die Bereichssperre für Ausnahmebehandlungen geeignet. Da beim Auftreten einer Ausnahme die lokalen Variablen gelöscht werden, wird auch die Bereichssperre gelöscht und somit das enthaltene Synchronisationsmittel freigegeben.

Der Mechanismus der Bereichssperre funktioniert nicht, wenn der Thread abnormal beendet wird.

3.2. Syslib

Ziel der Umgestaltung der Syslib ist das Herstellen von Threadsicherheit. Dazu wird das Modell der globalen Sperrung vorgestellt. Die Anforderung an dieses Modell ist, neben der Threadsicherheit, eine einfache Integration in die Syslib.

In einem zweiten Modell, der lokalen Sperrung, soll mehr auf die Leistungsfähigkeit geachtet werden.

3.2.1. Syslib und ThreadLib

Zuerst benötigt die Syslib die Fähigkeit, Synchronisationsmittel zu verwenden. Dazu wird eine neue Fabrikklasse *CSyLockFactory* erstellt. Sie erzeugt die benötigten Synchronisationsmittel aus der ThreadLib. Zu dieser internen Fabrikklasse existiert keine Schnittstelle. Erzeugt wird sie mit der globalen Funktion *getSyLockFactory()*. Alle Methoden der Fabrikklasse geben einen intelligenten Zeiger *PThrLock* zurück. Dieser verwaltet einen Zeiger auf die Schnittstelle *IThrLock*, welche die Basisklasse der Synchronisationsmittel ist. Die Methode *getSyslibLock()* erstellt ein globales Synchronisationsmittel. Dabei handelt es sich um einen rekursiven Mutex. Die Methode *getLockDummy()* erzeugt eine Dummy-Sperre. Die beiden Methoden *createMutex()* und *createRecMutex()* erzeugen jeweils einen einfachen und einen rekursiven Mutex.

Abbildung 3.14.: *CSyLockFactory*

3.2.2. Globale Sperrung

Grundlage

Die globale Sperrung soll den Zugriff auf die Syslib soweit einschränken, das nur ein Thread zu einem Zeitpunkt auf die Syslib zugreifen kann. Dies soll über ein globales Synchronisationsmittel gesteuert werden, welches aus der Klasse *CSyLockFactory* mit der Methode *getSyslibLock()* erzeugt werden kann.

Positionierung der globalen Sperrung

Die Syslib ist nach dem Entwurfsmuster der Brücke aufgebaut. Ziel ist die Integration der globalen Sperrung in die Brückenschicht. Die Aufgabe der Brückenklassen ist es, die Funktionsaufrufe an die Implementierungsschicht weiterzuleiten. Da alle Funktionsaufrufe nur über Brückenklassen erfolgen, ist dies ein geeigneter Punkt, ein Synchronisationsmittel einzufügen.

Beispiel:

```
iastring CSyNode::getName() const
{
  return m_impl->getName();
}
```

Dies ist die Brückenklasse *CSyNode*, welche die Methode *getName()* an *m_impl* weiterleitet. Die Variable *m_impl* ist ein intelligenter Zeiger auf das Implementierungsobjekt.

An dieser Stelle kann die Bereichssperre *CThrLocker* eingesetzt werden, um die Methode und somit den Funktionsaufruf zu schützen:

```
iastring CSyNode::getName() const
{
  CThrLocker locker(getSyLockFactory()->getSyslibLock());
  return m_impl->getName();
}
```

Integration in den intelligenten Zeiger

Der Aufwand, einen *CThrLocker* in alle vorhandenen Brückenklassen-Methoden einzufügen, wäre sehr hoch. Aus diesem Grund wird eine andere Möglichkeit gesucht, die globale Sperrung zu implementieren.

Die Alternative ist die Integration in den intelligenten Zeiger der Brückenklasse, der die Implementierungsklasse referenziert. Dazu soll die Sperrung in einem temporären Objekt

geschehen, das durch einen erweiterten intelligenten Zeiger erstellt wird. Das temporäre Objekt sperrt die globale Sperre und führt den Funktionsaufruf durch. Nach dessen Beendigung wird das temporäre Objekt gelöscht und gibt dabei die globale Sperre frei.

Die Klasse für den erweiterten intelligenten Zeiger ist *CSyLockPtr*. Das temporäre Objekt wird aus der Klasse *CSyLockHelper* erstellt.

CSyLockPtr

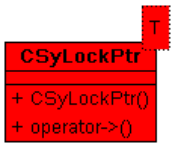


Abbildung 3.15.: *CSyLockPtr*

Der *CSyLockPtr* ist eine Ableitung der Klasse des intelligenten Zeigers *IaPtr*. Diese überschreibt die Methode *operator->()*. In der Basisklasse *IaPtr* gibt diese Methode den Zeiger auf das Objekt zurück, auf dem eine Funktion aufgerufen wird. Die Methode erstellt nun ein *CSyLockHelper* Objekt und gibt dieses zurück.

CSyLockHelper

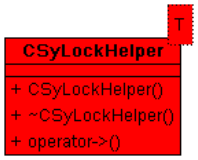


Abbildung 3.16.: *CSyLockHelper*

Die Hilfsklasse *CSyLockHelper* wird mit einem Zeiger und einem Synchronisationsmittel *PThrLock* erzeugt. Aufgabe der Klasse ist es, ähnlich wie eine Bereichssperre, das *PThrLock* Objekt im Konstruktor zu sperren und im Destruktor freizugeben. Außerdem besitzt die Klasse eine Methode *operator->()*. Wie bei einem intelligenten Zeiger gibt diese Methode den enthaltenen Zeiger zurück, um darauf einen Funktionsaufruf durchführen zu können.

Da das Objekt der Klasse *CSyLockHelper* nur solange existiert, bis die Funktion beendet ist, ist auch die Sperre nur für diese Zeit aktiv.

Fazit

Das vorgestellte Verfahren hat den Vorteil, dass nur in der Deklaration der Brückenklassen der Typ der Implementierungsklasse *m_impl* durch *CSyLockPtr* ersetzt werden muss. Da-

durch werden alle Methodenaufrufe, die über den erweiterten intelligenten Zeiger erfolgen, automatisch vor dem Funktionsaufruf gesperrt und danach freigegeben. Somit werden auch mögliche Fehlerquellen, wie das Vergessen des Sperrens oder Freigebens in den Brückenmethoden, ausgeschlossen.

Die Syslib ist somit threadsicher und es können keine Verklemmungen entstehen, da immer nur ein Thread die Syslib benutzen kann. Daraus folgt aber eine eingeschränkte Multithread-Fähigkeit, da nur ein Thread zu einem Zeitpunkt die Syslib benutzen kann und nur ein Prozessor belegt ist. Außerdem wird die Leistung durch den Mehraufwand des Synchronisationsverfahrens zusätzlich verringert. Wie sich dieser Mehraufwand auf die Leistung auswirkt, wird im Kapitel 5.2 analysiert.

3.2.3. Lokale Sperrung

Grundlage

Die lokale Sperrung soll parallele Zugriffe durch mehrere Threads auf die Syslib zulassen. Dazu muss die Synchronisation in der Implementierungsschicht vorgenommen werden. Als Synchronisationsmittel werden einfache und rekursive Mutexe verwendet, die aus der Klasse *CSyLockFactory* mit den Methoden *createMutex()* und *createRecMutex()* erzeugt werden.

Implementierungsschicht

Bevor die lokale Sperrung eingeführt werden kann, muss die interne Funktionsaufruf-Hierarchie betrachtet werden.

Die innere Struktur der Syslib ist nicht einheitlich. Dies betrifft insbesondere die Verwendung der internen Syslib-Objekte untereinander. Die Zugriffe erfolgen teilweise über Schnittstellen und teilweise direkt über die Implementierungsklassen.

Um eine einheitliche Funktionsaufruf-Hierarchie sicherzustellen, wird festgelegt, dass alle internen Verwendungen über Schnittstellen erfolgen müssen. Da die Syslib-Objekte als

threadsicher angenommen werden, dürfen interne Verwendungen der Schnittstellen nicht mit Synchronisationsmitteln gesichert werden. Diese Strategie verhindert das Eintreten einer Verklemmung, da die zweite Verklemmungsbedingung, die Halte- und Wartebedingung (siehe Kapitel 1.2.3), ausgeschlossen wird.

Da Syslib-Objekte auch interne Funktionalität von Implementierungsklassen benötigen, muss eine neue interne Schnittstellenschicht definiert werden.

Die internen Schnittstellen sind nach außen nicht sichtbar und bieten die benötigte interne Funktionalität an. Da die Funktionsaufrufe der internen Schnittstellen ebenfalls über die Brückenschicht erfolgen, stellen diese keine Verletzung der oben festgelegten Funktionsaufruf-Hierarchie dar.

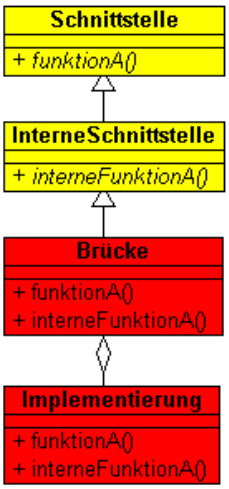


Abbildung 3.17.: Interne Schnittstelle

Um die internen Schnittstellen in Verbindung mit intelligenten Zeigern benutzen zu können, wird eine neue Klasse benötigt. Diese wird durch die Klasse *SyPtr* realisiert.

SyPtr

Der intelligente Zeiger *SyPtr* hat die gleiche Aufgabe, wie der bereits verwendete intelligente Zeiger *IaPtr*. Zusätzlich muss er die Konvertierung zwischen externen und internen Schnittstellen durchführen.

Einführung der lokalen Sperrung

Die Einführung der lokalen Sperrung ist schwieriger und zeitaufwendiger als bei der globalen Sperrung. Der Grund liegt darin, dass die Stellen, die geschützt werden müssen, noch nicht bekannt sind. Sie müssen erst durch genaues analysieren des Quelltextes identifiziert werden. Folgende Änderungen und Hinweise müssen beachtet werden:

- Interne Zugriffe[1] müssen geschützt werden.
- Externe Zugriffe[2] dürfen nur über *get()* und *set()* Methoden erfolgen. Interne Zugriffe sollten nach Möglichkeit auch über diese Methoden erfolgen.
- Die häufig verwendeten STL-Container sind nicht threadsicher und müssen an allen Verwendungsstellen geschützt werden.
- Öffentliche Methoden von Klassen dürfen keine Referenzen oder Zeiger zurückgeben.
- Bei konstanten Referenzen ist die Rückgabe möglich, da diese keine Veränderung des referenzierten Wertes erlauben. Dies sollte aber nur bei komplexen Objekten und nicht bei primitiven Datentypen gemacht werden. Der Benutzer sollte beachten, dass sich der Inhalt des referenzierten Objektes ändern kann und somit Inkonsistenzen auftreten können.
- Für das Schützen von Referenzen und Zeigern, die als Parameter an Funktionen übergeben werden, ist der Aufrufer verantwortlich.
- Private Methoden von Klassen dürfen Referenzen oder Zeiger zurückgeben. Diese müssen dann als nicht sichere Methoden behandelt werden.
- Als lokale Sperre kann der einfache Mutex verwendet werden. Es muss allerdings beachtet werden, dass dieser nicht zwei mal gesperrt werden kann.
- Intern verwendete Iteratoren, wie zum Beispiel STL-Iteratoren, müssen während des gesamten Iterationsvorganges geschützt werden.

[1] Eine Klasse benutzt ihre eigenen Attribute.
[2] Eine Klasse benutzt die Attribute anderer Klassen.

Fazit

Der Vorteil der lokalen Sperrung ist, dass nun mehrere Threads gleichzeitig auf die Syslib zugreifen können und somit die Multithread-Fähigkeit erhöht wird. Dies wird im Kapitel 5.3 genauer untersucht.

Allerdings ist die Implementierung mit einem hohen Aufwand verbunden, da eine genaue Analyse des Quelltextes notwendig ist.

3.2.4. Wechsel der Sperrstrategie

Nach der Integration der globalen Sperrung soll die Möglichkeit bestehen, die Sperre für Singlethread-Umgebungen zu deaktivieren. Dazu wird, anstatt des rekursiven Mutex, eine Dummy-Sperre verwendet. Somit kann der Synchronisationsaufwand vermieden werden, ohne die Struktur zu verändern.

Während der Integration der lokalen Sperrung bleibt die globale Sperrung aktiv. In den Methoden, die bereits durch lokale Sperrung geschützt sind, wird die globale Sperrung deaktiviert. Dies erlaubt ein schrittweises Einführen der lokalen Sperrung, ohne dabei die Threadsicherheit aufzugeben. Falls Fehler in der lokalen Sperrung vermutet werden, kann die globale Sperrung wieder aktiviert werden.

3.2.5. Komponenten außerhalb der Syslib

Diese Komponenten sind nicht Teil der Syslib. Da diese aber von der Syslib verwendet werden, müssen sie ebenfalls threadsicher gestaltet werden.

Referenzzählung

Das Problem der Referenzzählung wird gelöst, indem der Zugriff auf den Referenzzähler geschützt wird. Dazu wird ein rekursiver Mutex aus der ThreadLib verwendet.

Singleton

Singletons dürfen keinen Status mehr besitzen. Da dies in den Fabrikmethoden der Fall ist, stellen diese kein Problem dar. Singletons, die einen Status besitzen, müssen zu normalen Klassen umgewandelt werden.

Iteratoren

Intern verwendete Iteratoren können während des Iterierens durch Mutexe geschützt werden. Bei Iteratoren, die an den Benutzer gegeben werden, ist es nur möglich, die Iterator-Funktionen zu schützen. Für Veränderungen des Inhaltes des Iterators ist der Benutzer verantwortlich.

Systemfunktionen

Alle Systemfunktionen müssen geschützt werden. Somit wird sichergestellt, dass keine threadunsicheren Systemfunktionen verwendet werden. Dazu sollen alle Systemfunktionen, die in der Syslib verwendet werden, in einer zentralen Klasse gesammelt und dort geschützt werden. Im Rahmen der Diplomarbeit wird diese Konzeption nicht umgesetzt, sondern soll in zukünftigen Umstrukturierungen erfolgen.

3.3. Test-Framework für Leistungsanalyse

Durch die Einführung der globalen und lokalen Sperrung existieren nun Synchronisationsmittel, die Synchronisationsaufwand erzeugen und somit die Leistung beeinträchtigen. Um diese Beeinträchtigungen zu messen, sind Leistungstests auf C++-Ebene notwendig.

Ein geeignetes Test-Framework stellt das JUnit[3] Test-Framework dar. Allerdings sind Leistungstests in Java ungeeignet, da zu der C++-Ausführungszeit noch die Zeit für die SWIG-generierte Schicht hinzukommt.

3.3.1. Aufbau und Funktionsweise der Leistungstests

Für das Ausführen der Leistungstests wird das cppunit-Framework verwendet. Dies ist die C++-Umsetzung des Java JUnit Test-Frameworks.

Das cppunit-Framework wird bereits bei SAP für C++-Tests verwendet und wurde für die Diplomarbeit als Ausgangspunkt für die Leistungstests verwendet. Dabei wurden Änderungen durchgeführt, die eine Zeitmessung der einzelnen Tests erlauben.

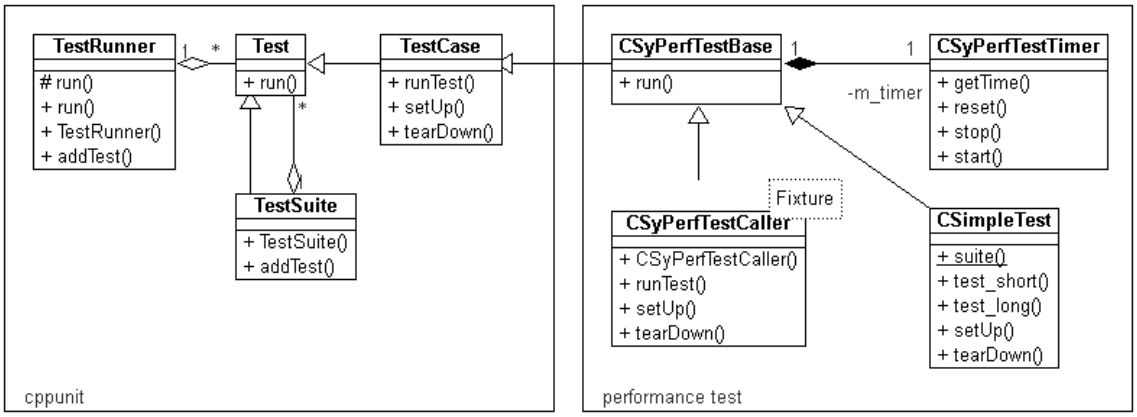


Abbildung 3.18.: Leistungstest Klassendiagramm

[3] JUnit - Test-Framework, Unterstützt die Entwicklung und Ausführung von Tests in Java

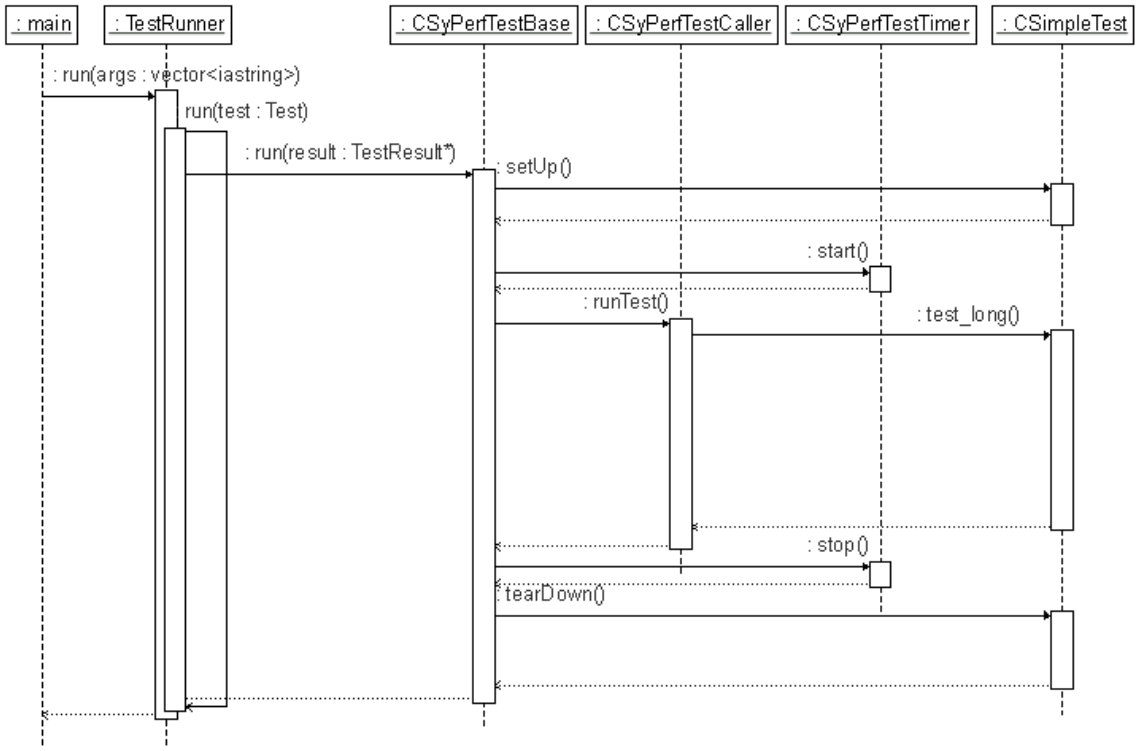


Abbildung 3.19.: Leistungstest Sequenz Diagramm

Es wird zwischen Tests und Testklassen unterschieden. Tests sind Testmethoden, die den zu testenden Programm-Code enthalten. Testklassen enthalten eine Ansammlung von mehreren Tests.
Einstiegspunkt ist die Klasse *TestRunner*. Dieser werden mit der Methode *addTest()* neue Testmethoden vom Typ *Test* hinzugefügt. Um Testmethoden aus Testklassen zu generieren, besitzt jede Testklasse eine Methode *suite()*. Diese erzeugt eine *TestSuite*, die alle Testmethoden einer Testklasse enthält. Da *TestSuite* eine Ableitung von *Test* ist, kann diese der Methode *addTest()* der Klasse *TestRunner* übergeben werden.

Beim Generieren von Testmethoden aus Testklassen wird das Objekt der Klasse *TestSuite* mit Objekten der Klasse *CSyPerfTestCaller<Fixture>* gefüllt. Dieser enthält einen Funktionszeiger auf die auszuführende Testmethode.

Um die Testmethoden auszuführen, wird die Methode *run(vector<iastring>)*[4] der Klasse *TestRunner* gerufen. Als Parameter wird dieser Methode ein *vector<iastring>* übergeben, der die Namen der auszuführenden Testklassen enthält. In dieser Methode werden durch Zeichenketten-Vergleich die Testmethoden herausgesucht und als Zeiger auf *Test* an die Methode *run(Test *)* der Klasse *TestRunner* übergeben. Diese Methode erzeugt ein *TextTestResult* Objekt, welches zum Speichern von Testergebnissen verwendet wird. Damit wird die Methode *run(TestResult *)* der Klasse *Test* aufgerufen. In der Ableitungshierarchie ist die Methode *run(TestResult *)* in der Klasse *CSyPerfTestBase* implementiert. Diese führt nun *setUp()*, *startTest()* und *tearDown()* aus. Aufgrund der Ableitungshierarchie werden diese Methoden aus der Klasse *CSyPerfTestCaller<Fixture>* ausgeführt. Die Methoden *setUp()* und *tearDown()* werden an die gleichnamigen Methoden der Testklasse *Fixture* delegiert. *startTest()* delegiert an die Testmethode.

Zur Zeitmessung wird die Klasse *CSyPerfTestTimer* verwendet. Ein Objekt dieser Klasse ist als Attribut in *CSyPerfTestBase* enthalten. Die Methode *start()* legt den Startzeitpunkt und die Methode *stop()* den Endzeitpunkt fest. Mit *getTime()* wird die verstrichene Zeit in Millisekunden zurückgegeben.

Um die Ausführungszeiten zu messen, wird die Zeitmessung vor der Methode *startTest()* gestartet und nach der Methode angehalten. Anschliessend werden die Zeiten in eine Logdatei ausgegeben. In der Zeitmessung werden die Methoden *setUp()* und *tearDown()* nicht mit erfasst, da diese Methoden der Initialisierung und Deinitialisierung des Tests dienen.

3.3.2. Testklasse *CTaskPerfTest*

Mit den Testmethoden der Testklasse *CTaskPerfTest* werden die Leistungstests durchgeführt. Dabei wird die Kopierfunktionalität der Dateisystem-Komponente der Syslib untersucht. Diese Komponente wurde für den Test ausgewählt, da das Kopieren ein wichtiger Bestandteil einer Installation ist.

[4] vector - STL-Container, speichert Daten in einem dynamischen Feld

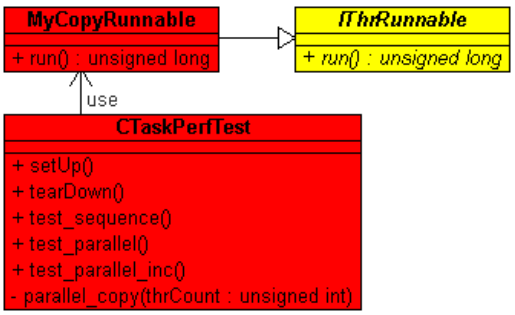


Abbildung 3.20.: *CTaskPerfTest*

Die Methode *setUp()* erzeugt einen Quellverzeichnisbaum mit Dateien. Alle erstellten Dateien, darunter der Quell- und Zielverzeichnisbaum, werden in der Methode *tearDown()* wieder gelöscht.

Die Methode *parallel_copy()* erzeugt Threads, deren Anzahl durch einen übergebenen Parameter bestimmt wird. Es wird eine Instanz der Klasse *MyCopyRunnable* erzeugt und an jeden erzeugten Thread als Zeiger übergeben. Somit arbeiten alle erzeugten Threads auf der gleichen *run()* Methode und mit den gleichen Attributen.

Die Testmethode *test_sequence()* startet den Test mit einem Thread. Dies entspricht dem Ausführen in einer Singlethread-Umgebung.

Mit der Testmethode *test_parallel()* wird der Test mit fünf Threads gestartet. Dies entspricht dem Ausführen in einer Multithread-Umgebung.

MyCopyRunnable

Diese Klasse ist eine Ableitung der Klasse *IThrRunnable* aus der ThreadLib. Sie enthält eine *run()* Methode, die den Einstiegspunkt eines Threads definiert.

Das Objekt der Klasse *MyCopyRunnable* wird mit einem Quellpfad und einem Zielpfad erzeugt. Beim Erstellen wird ein Syslib-Iterator für den Quellpfad angelegt und als Attribut im Objekt gespeichert.

Die Methode *run()* iteriert in einer Schleife mit dem Syslib-Iterator. Dieser repräsentiert ein Verzeichnis und enthält deswegen weitere Dateisystem-Objekte, die wiederum Dateien oder Verzeichnisse repräsentieren können. Auf jedem Dateisystem-Objekt wird eine Kopiermethode aufgerufen, die das Dateisystem-Objekt vom Quellpfad zum Zielpfad kopiert. Da die Kopiermethode auf dem Dateisystem-Objekt gerufen wird, ist der Dateityp für den Kopiervorgang nicht von Bedeutung.

3.3.3. Untersuchung des Aufwandes

Mit der Untersuchung des Aufwandes werden die Unterschiede der Ausführungszeiten von vier Syslib-Versionen analysiert. Diese Unterschiede entstehen durch die Implementierung der Synchronisationsmaßnahmen.

Es werden vier Versionen der Syslib untersucht:

1. Syslib mit globaler Sperrung
2. Syslib mit teilweise lokaler Sperrung
3. Syslib mit teilweise lokaler Sperrung (Dummy-Sperren)
4. Syslib ohne Sperren

Die Tests werden mit einem Thread ausgeführt, um eine Singlethread-Umgebung zu simulieren. Dies ist notwendig, da die Tests ohne Sperren und die mit Dummy-Sperren nicht korrekt in einer Multithread-Umgebung ausgeführt werden können.

Die Durchführung und Auswertung der Untersuchung befindet sich im Abschnitt 5.2.

3.3.4. Untersuchung der Multithread-Fähigkeit

Diese Untersuchung soll herausfinden, wie die Syslib ein Multiprozessorsystem ausgelastet. Dies soll über den Speedup ermittelt werden. Um diesen zu berechnen, werden die Ausführungszeiten einer sequenziellen und einer parallelen Version des Tests benötigt. Als sequenzielle Version wird der Test mit einem Thread ausgeführt. Für die parallele Version werden zwei und fünf Threads verwendet.

Die Durchführung und Auswertung der Untersuchung befindet sich im Abschnitt 5.3.

4. Implementierung

Dieses Kapitel beschreibt die Erstellung der ThreadLib und das Einführen der globalen und lokalen Sperrung in die Syslib.

4.1. ThreadLib

Bei SAP wird bereits eine C-Bibliothek verwendet, die Thread- und Synchronisationsmittel-Funktionalität bereitstellt. Die ThreadLib baut auf dieser C-Bibliothek auf und bildet eine Verschalung, die die Funktionalität in einer objektorientierten Klassenstruktur bereitstellt.

Die ThreadLib ist mit einer Schnittstellenschicht und einer Implementierungsschicht aufgebaut. Dabei bilden die beiden Fabrikklassen *IThrThreadFactory* und *IThrSyncFactory* die Einstiegspunkte, da nur sie ThreadLib-Objekte erzeugen können. Um dies sicherzustellen, werden die Konstruktoren aller Klassen *private* deklariert. Damit die Fabrikklassen diese dennoch erzeugen können, sind in allen Klassen die jeweiligen Fabrikklassen als *friend* deklariert.

Die Schnittstellenschicht wurde wie im Kapitel 3.1 umgesetzt. Dabei erhalten alle Schnittstellen, außer *IThrRunnable*, *IThrThreadBase* und *IThrLock*, eine Implementierungsklasse. Diese leiten die Funktionalität an die SAP-eigene C-Bibliothek weiter.

4.1.1. Bereichssperre

Die Funktionsweise der Bereichssperre basiert auf dem Konstruktions- und Destruktions-Verhalten von C++-Objekten. Aufgebaut sind sie mit der Hilfsklasse *CThrAutoLockerHlp*.

Folgende Synchronisationsmittel können mit Hilfe der Bereichsperre gesperrt werden:

Bereichssperre	Sperre	Beschreibung
CThrLocker	*PThrLock*	Basisklasse der Synchronisationsmittel
CThrMutexLocker	*PThrMutex*	einfacher Mutex
CThrRecMutexLocker	*PThrRecMutex*	rekursiver Mutex
CThrSemaphoreLocker	*PThrSemaphore*	Binär- und Zähl-Semaphore
CThrSharedLocker	*PThrRwLock*	Lese/Schreib-Sperre, Lese-Sperre
CThrExclusiveLocker	*PThrRwLock*	Lese/Schreib-Sperre, Schreib-Sperre

Tabelle 4.1.: Bereichssperre

CThrAutoLockerHlp

Diese Klasse ist eine Template-Klasse. Der Template-Typ *SyncType* ist das verwendete Synchronisationsmittel. Standardmäßig ruft die Template-Klasse im Konstruktor die *lock()* Methode und im Destruktor die *unlock()* Methode von *SyncType*. Dies ist bei *CThrLocker*, *CThrMutexLocker*, *CThrSemaphoreLocker* und *CThrRecMutexLocker* der Fall.

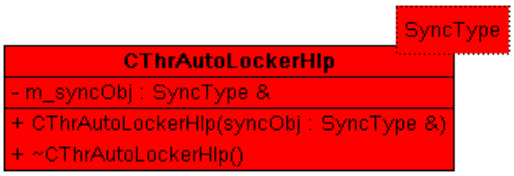


Abbildung 4.1.: *CThrAutolockerHlp*

Die Klassen *CThrSharedLocker* und *CThrExclusiveLocker* besitzen eine *unlock()* Methode, setzen aber in einer spezialisierten Template-Methode für die *lock()* Methode die Methoden *readLock()* und *writeLock()* ein.

Die Bereichssperren sind als *typedef* definiert, da es sich nicht um eigenständige Klassen handelt:

```
typedef CThrAutoLockerHlp<PThrMutex, true> CThrMutexLocker;
```

So kann eine Bereichssperre in einer Funktion eingesetzt werden:

```
void function()
{
  /* Mutex wird erstellt, ist nicht gesperrt */
  static PThrMutex mutex = getThrSyncFactory()->createMutex();
  /* Lokale Variable mutex wird automatisch gesperrt */
  CThrMutex locker(mutex);
  /* Geschützter Bereich */
}
/* Beim Verlassen der Funktion werden alle lokalen Variablen
   gelöscht, im Destruktor von locker wird der Mutex mutex
   freigegeben. */
```

Quelltext 4.1: Bereichssperre

4.2. Syslib

Es werden zwei Umgestaltungen in der Syslib vorgenommen. Die Erste erfolgt durch das Einführen der globalen Sperrung. Bei der Zweiten wird die lokale Sperrung eingeführt.

4.2.1. Globale Sperrung

Mit der globalen Sperrung soll verhindern werden, dass mehrere Threads gleichzeitig auf die Syslib zugreifen. Es wird nur einem Thread zu einem Zeitpunkt erlaubt, die Syslib zu betreten. Um dies zu gewährleisten, wird ein globaler rekursiver Mutex in die Brückenklassen integriert.

Die Brückenklassen besitzen in der Klassendefinition als Attribut einen Zeiger auf die Implementierungsklassen:

```
class CSyFile{
  /* Das Attribut vom Typ CSyFileImpl ist ein Zeiger auf das
     Implementierungsobjekt. */
private:
  CSyFileImpl *m_impl;
}
```

Quelltext 4.2: Vor der globalen Sperre

Dieses Attribut wird durch den erweiterten intelligenten Zeiger *CSyLockPtr<CSyFileImpl>* ersetzt, der die globale Sperrung automatisch vornimmt.

```
class CSyFile{
  /* Das Attribut ist nun ein intelligenter Zeiger,
        der ein Objekt der Klasse CSyFileImpl enthält. */
private:
  CSyLockPtr<CSyFileImpl> m_impl;
}
```

Quelltext 4.3: Nach der globalen Sperre

Die Ersetzung wurde mit einem Skript durchgeführt. Dieses sucht nach dem Attribut *Klasse *m_impl* und ersetzt es mit *CSyLockPtr<Klasse> m_impl*.

4.2.2. Lokale Sperrung

Im Kapitel 3.2.3 wurden Möglichkeiten beschrieben, wie eine lokale Sperrung eingeführt werden kann. Diese wurde für folgende Klassen durchgeführt:

- *CSyPath* - Ein Container für Dateisystempfade.
- *CSyNodeImpl* - Implementierungsklasse für Dateisystemknoten, wie zum Beispiel Dateien oder Verzeichnisse.
- *CSyFSPathAncestorIteratorImpl* - Implementierungsklasse für einen Dateisystempfad-Iterator.

Damit mussten, aufgrund von Klassenhierarchien und Abhängigkeiten, weitere Klassen threadsicher gestaltet werden:

- *CSyFileImpl* - Implementierungsklasse für Dateien
- *CSyDirectoryImpl* - Implementierungsklasse für Verzeichnisse
- *CSyLinkImpl* - Implementierungsklasse für Links unter UNIX
- *CSySpecialFileImpl* - Implementierungsklasse für Spezielle Dateitypen unter UNIX
- *CSyFSPathImpl* - Implementierungsklasse für Dateisystempfade

Änderung in der Klasse *CSyPath*

In der Klasse *CSyPath* wurden die meisten Änderungen vorgenommen. Diese Klasse ist auf den STL-Container *deque* aufgebaut.

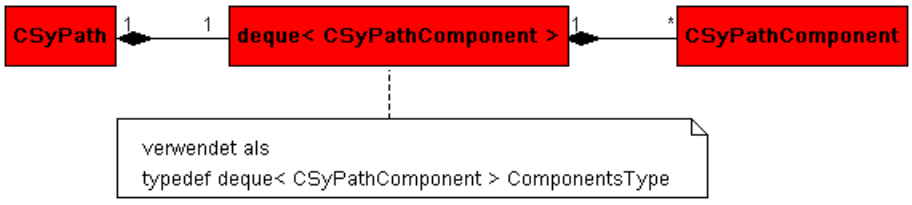


Abbildung 4.2.: *CSyPath*

An vielen Code-Stellen wird das Attribut *ComponentsType m_components* direkt verwendet. Dies wird durch eine neue threadsichere Methode *getComponents()* ersetzt.

```
CSyPath::ComponentsType CSyPath::getComponents() const
{
  CThrLocker locker(m_lock);
  return m_components;
}
```

Quelltext 4.4: *getComponents()*

Die Verwendung dieser Methode führt jedoch zu einem weiteren Problem. Einige externe Methoden, die von der Klasse *CSyPath* als *friend* Methoden deklariert wurden, greifen direkt auf das Attribut *m_components* zu und erzeugen mittels *begin()* und *end()* Start- und End-Iteratoren.

```
bool equalCaseSensitive(const CSyPath& lhs, const CSyPath& rhs)
{
  /* ... */
  if (lhs.m_components.size() != rhs.m_components.size())
    {return false;}
  return equal(lhs.m_components.begin(), lhs.m_components.end(),
    rhs.m_components.begin(), &equalCaseInsensitiveCmp);
}
```

Quelltext 4.5: *equalCaseSensitive()* nicht threadsicher

Ein einfaches Ersetzen von *m_components* nach *getComponents()* ist in diesem Fall nicht möglich, da *getComponents()* eine neue Instanz von dem Attribut erzeugt. Dadurch würden Start- und End-Iteratoren erzeugt werden, die auf unterschiedliche Container verweisen. Die Lösung ist das Zwischenspeichern des Attributs als lokale Variable:

```
bool equalCaseSensitive(const CSyPath& lhs, const CSyPath& rhs)
{
  /* ... */
  CSyPath::ComponentsType lhsComp = lhs.getComponents();
  CSyPath::ComponentsType rhsComp = rhs.getComponents();
  if (lhsComp.size() != rhsComp.size())
    {return false;}
  return equal(lhsComp.begin(), lhsComp.end(),
    rhsComp.begin(), &equalCaseSensitiveCmp);
}
```

Quelltext 4.6: *equalCaseSensitive()* threadsicher

Es werden nur threadsichere Methoden verwendet und auf lokalen Variablen gearbeitet. Somit ist die Methode threadsicher. Nachteil dieser Umgestaltung ist ein geringer Leistungsverlust, da die Attribute kopiert und zwischengespeichert werden.

Änderung in der Klasse *CSyNodeImpl*

In der Klasse *CSyNodeImpl* befindet sich ein weiteres Beispiel für einen zu schützenden Bereich. An dieser Stelle muss der Status der Klasse während des Durchlaufens eines kritischen Abschnittes konstant gehalten werden.

Dazu wird folgende Methode betrachtet:

```
iastring CSyNodeImpl::getName() const
{
  iastring result(m_path.toString());
  int unused1; iastring unused2;
  if (m_path.canSplitIntoParentAndFile()){
    m_path.splitIntoParentAndFile().second.splitFileName(result,
       unused1, unused2);
  }
  return result;
}
```

Quelltext 4.7: *getName()* nicht threadsicher

Diese Methode berechnet aus dem Attribut *CSyPath m_path* den Namen des Pfades. Dazu wird die Methode *splitIntoParentAndFile()* verwendet, um den Pfad in den Elternpfad und den Namen zu trennen. Vorher wird mit *canSplitIntoParentAndFile()* getestet, ob eine solche Trennung möglich ist. Dies kann in einer Multithread-Umgebung zu Problemen führen. Ein Thread kann die Methode *canSplitIntoParentAndFile()* erfolgreich passieren, während ein anderer Thread das Attribut *m_path* so ändert, dass die obige Methode fehlschlagen würde. Um dieses Problem zu vermeiden, wird ein *CThrLocker* in einem Block eingesetzt:

```
iastring CSyNodeImpl::getName() const
{
  iastring result(m_path.toString());
  int unused1; iastring unused2;
  {
    CThrLocker locker(m_lock);
        /* Ab hier ist die Bereichssperre aktiv */
    if (m_path.canSplitIntoParentAndFile()){
      m_path.splitIntoParentAndFile().second.splitFileName(result,
          unused1, unused2);
    }
  }     /* Bis hier ist die Bereichssperre aktiv */
  return result;
}
```

Quelltext 4.8: *getName()* threadsicher

Diese Umstellung verhindert eine Änderung des Attributs *m_path* während des kritischen Abschnittes. Damit die Methode endgültig threadsicher ist, werden alle Code-Stellen, die das Attribut *m_path* manipulieren, mit *m_lock* geschützt.

4.2.3. Wechsel der Sperrstrategie

Um die globale Sperrung für bestimmte Methoden, die bereits durch lokale Sperrung geschützt sind, zu umgehen, wird die Klasse *CSyLockPtr* mit der Methode *localLocked()* erweitert. Diese gibt ein Objekt der Klasse *CSyLockHelper* zurück, welches anstatt der globalen Sperre eine Dummy-Sperre enthält. Somit wird die globale Sperrung für bestimmte Methoden deaktiviert.

Ein Mechanismus, der die globale Sperrung vollständig aktivieren und deaktivieren kann, wurde in dieser Diplomarbeit nicht implementiert.

4.2.4. Komponenten außerhalb der Syslib

Folgende Änderungen in den untersuchten Komponenten wurden durchgeführt:

Referenzzählung

Es wurde ein rekursiver Mutex aus der ThreadLib verwendet, um den Zugriff auf den Referenzzähler zu synchronisieren.

Singleton

Die Fabrikklassen besitzen keinen Status und sind threadsicher. Weitere Änderungen an den Singletons wurden nicht durchgeführt.

Iteratoren

Die Synchronisation der Iteratoren wurde nicht im Rahmen der Diplomarbeit durchgeführt.

Systemfunktionen

Eine zentrale Klasse für Systemfunktionen wurde nicht erstellt. Es war allerdings beim Einführen der lokalen Sperrung notwendig, die Systemfunktionen für die Kopierfunktionalität zu schützen.

5. Leistungstest

In diesem Kapitel werden die, im Kapitel 3.3 entworfenen, Leistungstests durchgeführt und ausgewertet. Die Tests werden aus der Testklasse *CTaskPerfTest* mit der Testmethode *test_parallel()* durchgeführt.

5.1. Testkonfiguration

Auf folgenden Systemen werden die Tests durchgeführt:

Abkürzung	Beschreibung
ntintel	Windows XP SP2, 32Bit Intel Core 2 Duo 3GHz, 2GB RAM
linuxx86_64	SUSE Linux Enterprise Server 10, 64 Bit 2 x Intel Core 2 Duo 2.4GHz, 4GB RAM
hp_64	HP-UX, 64 Bit 9000/800/rp3440, 8GB RAM
sunx86_64	SunOS, 64 Bit 2 x AMD Opteron Processor 1.8GHz, 8GB RAM

Tabelle 5.1.: Testsysteme

Beim Kopiervorgang werden 100 Verzeichnisse mit jeweils 10 Dateien erstellt. Kopiert wird auf der Verzeichnisebene, indem ein Thread jeweils ein Verzeichnis kopiert.

Die Tabellen mit den Ausführungszeiten befinden sich im Anhang B.1

5.2. Untersuchung des Aufwandes

In dieser Untersuchung werden vier Syslib-Versionen anhand der Ausführungszeiten der Tests verglichen. Dabei sollen die Unterschiede der Ausführungszeiten eine Aussage über den Aufwand der Synchronisationsmaßnahmen treffen.

Getestet wird in einer Singlethread-Umgebung.

5.2.1. Durchführung

Folgende Syslib-Versionen werden getestet:

1. Syslib mit globaler Sperrung: Enthält eine globalen Sperre
2. Syslib mit teilweise lokaler Sperrung: Enthält globale Sperrung, wobei einige Methoden anstelle der globale Sperren durch lokale Sperren synchronisiert werden
3. Syslib mit teilweise lokaler Sperrung (Dummy-Sperre): Wie 2., enthält aber Dummy-Sperren
4. Syslib ohne Sperren: Eine Syslib-Version vor der Einführung der Sperren

5.2.2. Auswertung

In diesem Diagramm sind die Testergebnisse der vier Plattformen Windows (ntintel), Linux (linuxx86_64), HP-UX (hp_64) und SunOS (sunx86_64) dargestellt. Der Kopiertest wurde mit 1 KByte-Dateien durchgeführt. Dabei wurde aus jeweils 20 Testläufen der Mittelwert gebildet.

Zur Darstellung sind die Ausführungszeiten auf dem Wert der lokalen Sperrung skaliert und mit 100 multipliziert. Somit sind die Verhältnisse der Ausführungszeiten zwischen den Syslib-Versionen auf den jeweiligen Plattformen anschaulich dargestellt.

Abbildung 5.1.: Legende, vier Syslib-Versionen

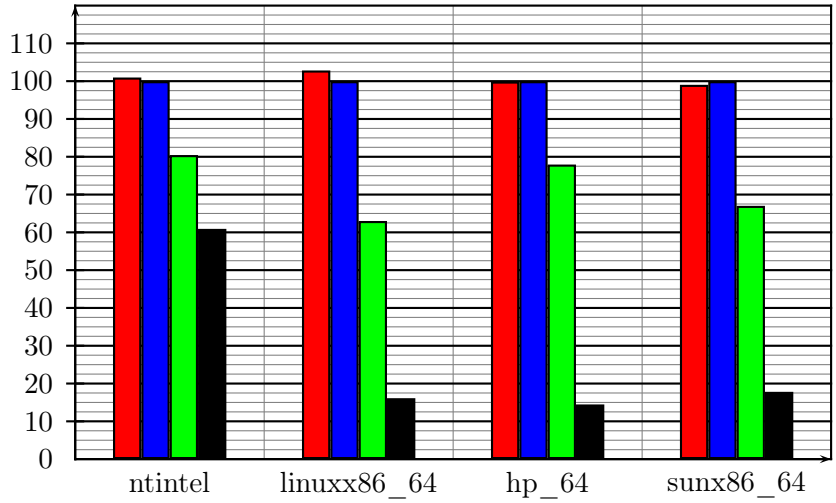


Abbildung 5.2.: Ergebnisdiagramm für 1 Thread, 1KByte Datei, skaliert

Das Diagramm zeigt einen deutlichen Unterschied der Ausführungszeit zwischen der Syslib ohne Sperren und der Syslib mit globaler bzw. lokaler Sperrung. Besonders auf den Unix-Plattformen verlängert sich die Ausführungszeit um den Faktor 6.6.

Einflussfaktoren für die Verlängerung der Ausführungszeit sind das Einführen des Modells der globalen Sperrung, der lokalen Sperrung und der internen Schnittstellen. Das Betriebssystem erzeugt, durch das Sperren und Freigeben von Synchronisationsmitteln, ebenfalls Aufwand.

Der Unterschied der Ausführungszeiten zwischen der Syslib ohne Sperren und der Syslib mit Dummy-Sperren ist ebenfalls bemerkenswert. Trotz des Wegfalls des Synchronisationsaufwandes des Betriebssystems ist der Unterschied der Ausführungszeiten noch deutlich vorhanden. Dies ist auf die Änderung der internen Struktur der Syslib zurückzuführen.

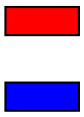
Zwischen der globalen und lokalen Sperrung gibt nur geringe Unterschiede der Ausführungszeiten, da nur der Synchronisationsaufwand des Betriebssystems, in den Brückenklassen, wegfällt.

5.3. Untersuchung der Multithread-Fähigkeit

5.3.1. Durchführung

Die Multithread-Fähigkeit der Syslib wird über den Speedup gemessen. Um diesen zu bestimmen, werden die Ausführungszeiten von einem sequenziellen Test und einem parallelen Test benötigt. Für die Ausführungszeiten des sequenziellen Tests werden die Ergebnisse aus der vorhergehenden Untersuchung der lokalen Sperrung benutzt, da diese bereits mit einem Thread ausgeführt wurden. Der parallele Test wird mit zwei und fünf Threads ausgeführt, wobei die Syslib-Versionen der globalen und der lokalen Sperrung verwendet werden. Es werden wieder 1 KByte Dateien kopiert.

5.3.2. Auswertung



Globale Sperrung

Lokale Sperrung

Abbildung 5.3.: Legende, zwei Syslib-Versionen

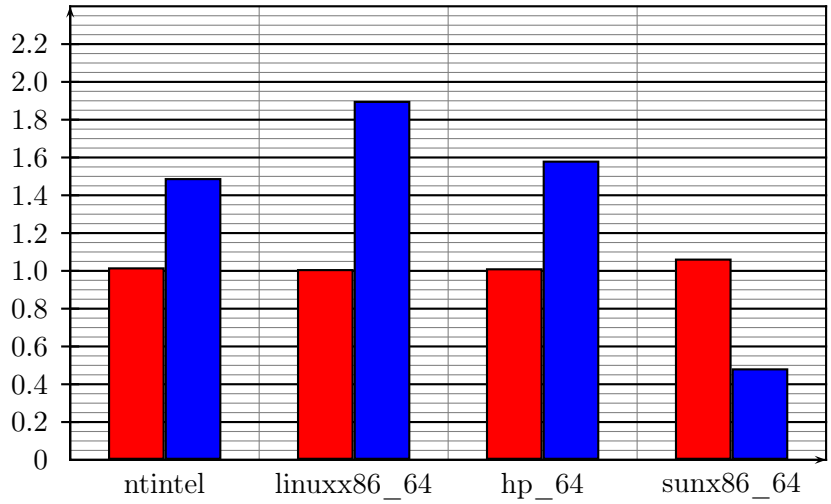


Abbildung 5.4.: Ergebnisdiagramm für 2 Threads, 1KByte Datei, Speedup

Der Speedup ist bei der lokalen Sperrung auf Linux mit 1.9 am Größten. Dies ist auf das Testsystem zurückzuführen, da dieses 2 Prozessoren mit jeweils 2 Kernen besitzt. Somit kann diese Plattform das Multithreading besonders gut umsetzen.

Mit einem Speedup von nur 0.48, kommt es auf der SunOS-Plattform zu einer Verschlechterung durch den Einsatz von Multithreading.

Wie erwartet, liegt der Speedup für die globale Sperrung auf allen Plattformen bei 1. Somit bringt der Einsatz vom Multithreading keine Vorteile.

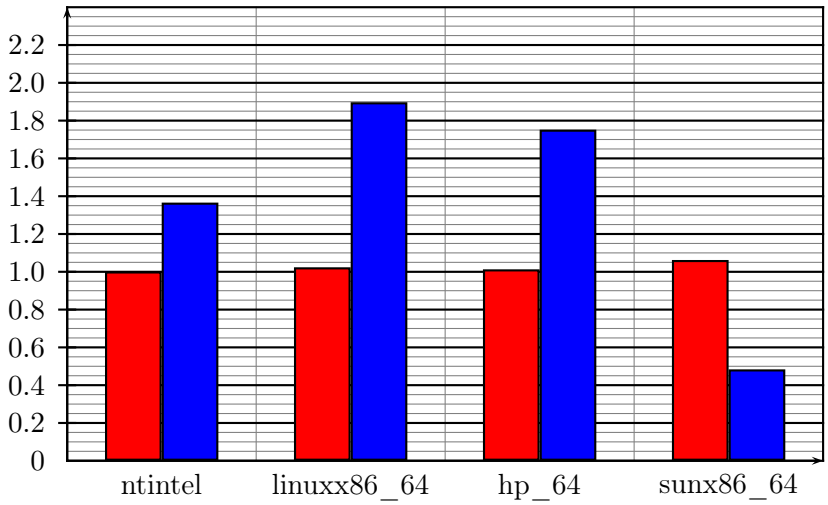


Abbildung 5.5.: Ergebnisdiagramm für 5 Threads, 1KByte Datei, Speedup

Durch die Ausführung mit fünf Threads verringert sich der Speedup auf Windows von 1.49 auf 1.37. Dies ist vermutlich auf einen Leistungseinbruch der Festplatte zurückzuführen, der bei vielen Zugriffen entsteht.

Auf HP-UX verbessert sich der Speedup von 1.58 auf 1.75. Durch die hohe Ausführungszeit und den somit vermutlich geringen Festplattendurchsatz, sind auf HP-UX noch Verbesserungen des Speedups durch die Ausführung mehrerer Threads möglich.

6. Zusammenfassung

6.1. Entwicklungsstatus

Der aktuelle Stand der Syslib ist threadsicher. Dies ist durch die globale und lokale Sperrung gewährleitet. Somit wurde das Ziel der threadsicheren Gestaltung der Syslib erreicht.

Eine Erhöhung der Leitungsfähigkeit der Syslib wurde nicht erreicht. Durch die threadsichere Gestaltung wurde sogar die Leistung eingeschränkt.

6.2. Ergebnisse der Tests

Die Untersuchungen ergeben einen unerwartet hohen Synchronisierungsaufwand und somit eine wesentliche Erhöhung der Ausführungszeiten. Auch das Ausführen mit mehreren Threads kann die Ausführungszeiten der threadsicheren Syslib-Versionen, im Verleich zur Syslib-Version ohne Sperren, nicht verbessern.

Bei den durchgeführten Tests hat die Leistung der Festplatte einen wesentlichen Einfluss auf die Ausführungszeiten. Deshalb wird vermutet, dass dieser Einfluss den Parallelierungsgrad der Kopiertests einschränkt. Dies könnte den schlechten Speedup für den parallelen Test auf der SunOS-Plattform erklären.

6.3. Ausblick

Weitere Untersuchungen mit Tests aus den verschiedenen Komponenten der Syslib sind notwendig, um die Ursache des erzeugten Synchonisationsaufwandes genauer zu analysieren.

Um andere Komponenten der Syslib zu untersuchen, ist die vollständige Implementierung der lokalen Sperrung notwendig.

A. Erklärung zur selbständigen Anfertigung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Bearbeitungsort, Datum	Unterschrift
.....	.....

B. Anhang

B.1. Tabellen

Dies sind die Tabellen der Testergebnisse der durchgeführten Leistungstests.

Sequenzieller Test

	ntintel	linuxx86_64	hp_64	sunx86_64
Globale Sperre	100.97	102.83	99.91	99.03
Lokale Sperre	100.00	100.00	100.00	100.00
Lokale Sperre (Dummy)	80.45	63.00	77.94	67.02
Keine Sperre	60.92	16.12	14.47	17.81

Tabelle B.1.: Ergebnistabelle für 1 Thread, 1KByte Datei, skaliert

	ntintel	linuxx86_64	hp_64	sunx86_64
Globale Sperre	14480	11084	97490	13123
Lokale Sperre	14341	10779	97582	13251
Lokale Sperre (Dummy)	11538	6791	76054	8880
Keine Sperre	8737	1738	14120	2360

Tabelle B.2.: Ergebnistabelle für 1 Thread, 1KByte Datei, in Millisekunden

Paralleler Test, zwei Threads

	ntintel	linuxx86_64	hp_64	sunx86_64
Globale Sperre	1.02	1.01	1.01	1.07
Lokale Sperre	1.49	1.90	1.58	0.48

Tabelle B.3.: Ergebnistabelle für 2 Threads, 1KByte Datei, Speedup

	ntintel	linuxx86_64	hp_64	sunx86_64
Globale Sperre	14618	10955	97520	13228
Lokale Sperre	9352	5842	62103	29767

Tabelle B.4.: Ergebnistabelle für 2 Threads, 1KByte Datei, in Millisekunden

Paralleler Test, fünf Threads

	ntintel	linuxx86_64	hp_64	sunx86_64
Globale Sperre	1.00	1.02	1.01	1.06
Lokale Sperre	1.37	1.90	1.75	0.48

Tabelle B.5.: Ergebnistabelle für 5 Threads, 1KByte Datei, Speedup

	ntintel	linuxx86_64	hp_64	sunx86_64
Globale Sperre	14863	10804	97604	13262
Lokale Sperre	10208	5850	56121	29811

Tabelle B.6.: Ergebnistabelle für 5 Threads, 1KByte Datei, in Millisekunden

B.2. Quelltext

Der Quelltext des erstellten Test-Frameworks ist auf der beiliegenden CD-ROM zu finden. Die ThreadLib und die Änderungen in der Syslib sind nicht im Anhang enthalten. Diese befinden sich bei SAP.

Im folgenden Abschnitt befindet sich der Quelltext der Testklasse *CTaskPerfTest*.

syxxpfttask.hpp

Quelltext B.1: *CTaskPerfTest* Header-Datei

```
1  #ifndef SYXXPFTTASK_HPP_
2  #define SYXXPFTTASK_HPP_
3
4  #include "iaxxglbdef.hpp"
5  USING_NAMESPACE_STD
6  #include "syxxpftbase.hpp"
7  #include "syxxifsmgt.hpp"
8
9  class ILogBook;
10
11 class CTaskPerfTest : public CSyPerfTestBase
12 {
13 public:
14         CTaskPerfTest(const iastring &name) : CSyPerfTestBase(name)
               {}
15         static Test* suite ();
16
17         void setUp(ILogBook* lgb);
18         void tearDown(ILogBook* lgb);
19
20         void test_sequence(ILogBook* lgb);
21         void test_parallel(ILogBook* lgb);
22         void test_parallel_inc(ILogBook* lgb);
23 private:
```

```
24          void parallel_copy(unsigned int thrCount);
25
26          PSyFileSystemMgt m_fsmgt;
27          const static iastring m_dirpath;
28          const static iastring m_dirpath2;
29          const static iastring m_filename;
30  };
31
32  #endif /* SYXXPFTTASK_HPP_ */
```

syxxpfttask.cpp

Quelltext B.2: *CTaskPerfTest* Quelldatei

```
1  #include "iaxxglbdef.hpp"
2  #include <queue>
3  USING_NAMESPACE_STD
4
5  #include "syxxpfttask.hpp"
6  #include "syxxpftcal.hpp"
7  #include "iaxxutsuit.hpp"
8  #include "iaxxithrrunnable.hpp"
9  #include "iaxxthrcreate.hpp"
10 #include "iaxxithr.hpp"
11 #include "syxxifspath.hpp"
12 #include "syxxsyfact.hpp"
13 #include "syxxifile.hpp"
14 #include "syxxidir.hpp"
15 #include "syxxinodit.hpp"
16 #include "syxxitime.hpp"
17 #include "syxxisize.hpp"
18 #include "syxxinodti.hpp"
19 #include "syxxifstrm2.hpp"
20 #include "iaxxexbase.hpp"
21
22 const iastring CTaskPerfTest::m_dirpath =L"testdir" ;
23 const iastring CTaskPerfTest::m_dirpath2 =L"testdircopy" ;
24 const iastring CTaskPerfTest::m_filename =L"testfile" ;
25
26 // if defined, time measurement and reporting is activ
27 #define TASK_PERF_TEST_TIME_MEASURE
28
29 #ifdef TASK_PERF_TEST_TIME_MEASURE
30 void writeSimplePerfLog(iastring filename, iastring string);
31 #endif
```

```
32
33 class MyCopyRunnable : public IThrRunnable
34 {
35 private:
36         PSyFSPath m_sourceDirPath;
37         PSyFSPath m_targetDirPath;
38         PSyFileSystemMgt m_fsmgt;
39         PThrLock m_lock;
40         PSyNodeIterator m_nodeIterator;
41
42         unsigned long run()
43         {
44                 // weak check if is done
45                 while(!m_nodeIterator->isDone())
46                 {
47                         PSyNode node;
48                         {
49                                 CThrLocker locker(m_lock);
50                                 // guarded check if is done
51                                 if(m_nodeIterator->isDone())
52                                         continue;
53                                 node = m_nodeIterator->get();
54                                 m_nodeIterator->next();
55                         }
56                         iastring filename = node->getFullNameNoPath
                                ();
57                         node->copy(m_targetDirPath->concat(m_fsmgt
                                ->getFSPath(filename)));
58                 }
59                 iacerr<<L"end:␣"<<getThrThreadFactory()->
                        createCurrentThread()->dumpId()<<endl;
60                 return 0;
61         }
62 public:
```

```
63          MyCopyRunnable(PSyFSPath sourceDirPath,PSyFSPath
               targetDirPath):
64                  m_sourceDirPath(sourceDirPath),
65                  m_targetDirPath(targetDirPath),
66                  m_fsmgt(getSyslibFactory999()->getFileSystemMgt()),
67                  m_lock(getThrSyncFactory()->createMutex())
68          {
69                  m_nodeIterator =  m_fsmgt->getDirectory(
                       sourceDirPath)->getChildNodes(L"*",false);
70          }
71  };
72
73  void CTaskPerfTest::setUp(ILogBook* lgb)
74  {
75          m_fsmgt = GET_SYSLIB_FUNCTION_NAME_VERSION_DEPENDEND(
               getSyslibFactory)()->getFileSystemMgt();
76          tearDown(NULL);
77          m_fsmgt->createDirectory(m_fsmgt->getFSPath(m_dirpath2));
78
79          m_fsmgt->createDirectory(m_fsmgt->getFSPath(m_dirpath));
80
81          PSyFile testFile = m_fsmgt->createFile(
82                          m_fsmgt->getFSPath(m_dirpath)->concat(
                                m_fsmgt->getFSPath(L"testfile")));
83          PSyFileStream2 stream = testFile->getFileStream2(ISyFile::
               WRITE);
84          // create a 1024 Byte file
85          for(int i=0;i!=64;++i)
86                  stream->putString(L"0123456789abcdef");
87          stream->close();
88
89          // create 100 directories
90          for (int i = 0; i != 100; ++i)
91          {
```

```
92                  iastring dirstr = m_dirpath + stringFromSLong(i);
93                  PSyFSPath dirpath = m_fsmgt->getFSPath(m_dirpath)->
                        concat(
94                                      m_fsmgt->getFSPath(dirstr));
95                  m_fsmgt->createDirectory(dirpath);
96                  // create 10 files per directory
97                  for (int j = 0; j != 10; ++j)
98                  {
99                          iastring filestr = m_filename +
                            stringFromSLong(j);
100                         PSyFSPath filepath = dirpath->concat(
                            m_fsmgt->getFSPath(filestr));
101
102                         testFile->copy(filepath);
103                 }
104         }
105         testFile->remove();
106 }
107
108 void CTaskPerfTest::tearDown(ILogBook* lgb)
109 {
110         // remove testdir and testdircopy
111         PSyFSPath path;
112         path = m_fsmgt->getFSPath(m_dirpath);
113         if (m_fsmgt->isExisting(path, ISyNode::DIRECTORY))
114         {
115                 PSyDirectory dir = m_fsmgt->getDirectory(path);
116                 dir->remove();
117         }
118         path = m_fsmgt->getFSPath(m_dirpath2);
119         if (m_fsmgt->isExisting(path, ISyNode::DIRECTORY))
120         {
121                 PSyDirectory dir = m_fsmgt->getDirectory(path);
122                 dir->remove();
```

```
123         }
124 }
125
126 void CTaskPerfTest::parallel_copy(unsigned int thrCount)
127 {
128         if(thrCount <= 0)
129                 return;
130         // create source and destination path
131         PSyFSPath source = m_fsmgt->getFSPath(m_dirpath);
132         PSyFSPath dest = m_fsmgt->getFSPath(m_dirpath2);
133         PThrRunnable mytask = new MyCopyRunnable(source, dest);
134
135         // create and start threads
136         vector<PThrThread> thr;
137         for (unsigned int i = 0; i < thrCount; i++)
138         {
139                 thr.push_back(getThrThreadFactory()->createThread(
                    mytask));
140                 thr.at(i)->start();
141         }
142         // wait for threads to finish
143         for (unsigned int i = 0; i < thr.size(); i++)
144         {
145                 thr.at(i)->join();
146         }
147 }
148 // test parallel_copy with increasing thread count
149 void CTaskPerfTest::test_parallel_inc(ILogBook* lgb)
150 {
151         const iastring fileName = L"CTaskPerfTest--
            test_parallel_inc.log";
152         const iastring fileNameLoad = L"CTaskPerfTest--
            test_parallel_inc_load.log";
153         const unsigned int count = 20;
```

```
154 #ifdef TASK_PERF_TEST_TIME_MEASURE
155         CSyPerfTestTimer timer;
156 #endif
157         for(unsigned int i=1;i!=count+1;++i)
158         {
159                 setUp(NULL);
160
161 #ifdef TASK_PERF_TEST_TIME_MEASURE
162                 timer.start();
163 #endif
164
165                 parallel_copy(i);
166
167 #ifdef TASK_PERF_TEST_TIME_MEASURE
168                 timer.stop();
169 #endif
170
171                 tearDown(NULL);
172
173 #ifdef TASK_PERF_TEST_TIME_MEASURE
174                 writeSimplePerfLog(fileName,stringFromSLong(timer.
                        getTime()));
175                 writeSimplePerfLog(fileNameLoad,stringFromDouble(
                        timer.getCPULoadProcess()));
176                 timer.reset();
177 #endif
178         }
179 }
180 // sequence test with one thread
181 void CTaskPerfTest::test_sequence(ILogBook* lgb)
182 {
183         parallel_copy(1);
184 }
185 // parallel test with 5 threads
```

```
186 void CTaskPerfTest::test_parallel(ILogBook* lgb)
187 {
188         parallel_copy(5);
189 }
190
191 Test*
192 CTaskPerfTest::suite()
193 {
194         TestSuite* testSuite = new TestSuite(iaS("CTaskPerfTest"));
195         testSuite->addTest(new CSyPerfTestCaller<CTaskPerfTest> (
              iaS("CTaskPerfTest::test_parallel_inc"),
196         &CTaskPerfTest::test_parallel_inc));
197         testSuite->addTest(new CSyPerfTestCaller<CTaskPerfTest> (
              iaS("CTaskPerfTest::test_sequence"),
198         &CTaskPerfTest::test_sequence));
199         testSuite->addTest(new CSyPerfTestCaller<CTaskPerfTest> (
              iaS("CTaskPerfTest::test_parallel"),
200         &CTaskPerfTest::test_parallel));
201         return testSuite;
202 }
203
204 #ifdef TASK_PERF_TEST_TIME_MEASURE
205 void writeSimplePerfLog(iastring filename, iastring string)
206 {
207         static PThrMutex s_mutex = getThrSyncFactory()->createMutex
              ();
208         CThrMutexLocker locker(s_mutex);
209         fstream file;
210         file.open(toMbstring(filename).c_str(), fstream::out |
              fstream::app);
211         file << toMbstring(string) << endl;
212         file.close();
213 }
214 #endif
```

C. Literaturverzeichnis

[Han95] Hansen, Olav: *Leistungsanalyse Paralleler Programme* - Spektrum Akademischer Verlag, 1995

[Sch00] Schneider, Uwe; Werner, Dieter: *Taschenbuch der Informatik* - 3. Aufl. - Fachbuchverlag Leipizg, 2000

[Car06] Carver, Richard H.; Tai, Kuo-Chung: *Modern Multithreading* - Wiley, 2006

[Goe07] Goetz, Brian; Peilers, Tim; Bloch, Joshua; ...: *Java Conurrency in Practice* - Addison Wesley, 2007

[Sch07] Schmidt, Dougles; Stal, Michael; Rohnert, Hans; ...: *Pattern-Oriented Software Architecture, Pattern for Concurrent and Networked Objects, Volume 2* - Wiley, 2007

[Rei07] Reinders, James: *Intel Threading Building Blocks* - O´Reilly, 2007

[SAP] SAP: *SAPinst Developers Guide* - 2004

[SWIG] SWIG Executive Summary. URL: <*http://www.swig.org/exec.html*>, letzte Änderung 27.09.2007