

# Class Coroutine

Armin M. Warda

27. September 1995

Implementationsbeschreibung einer  
C++-Klasse für SIMULA-ähnliche Coroutinen  
mit Checkpoint und Rollback

## 1 Einleitung und Hintergrund

Dieses Kapitel beschreibt die Implementierung<sup>1</sup> eines SIMULA-ähnlichen Coroutinen-Konzeptes als C++-Klasse mit Hilfe von *leichtgewichtigen Prozessen* (*Light Weight Processes, LWP*). Im Gegensatz zu anderen Implementierungen läßt sich der Zustand einer solchen Coroutine sichern (*Checkpoint*) und später wiederherstellen (*Rollback*), während die Coroutine zwischenzeitlich *spekulativ* fortgesetzt worden sein kann. Die Operationen Checkpoint und Rollback sind (unter anderem) unentbehrlich für die Realisierung von optimistischen Protokollen zur parallelen Prozeß-orientierten Simulation.

Die Implementierung unterteilt sich in zwei Abschnitte: zunächst wird ein LWP-System realisiert, dessen Implementierung in hohem Maße Rechnerarchitektur- und Betriebssystem-abhängig ist. Das LWP-System verdeckt allerdings bereits alle Plattform-spezifischen Details der Implementierung, sodaß die auf dem LWP-System basierende Implementierung der C++-Klasse „Coroutine“ von der konkreten Rechnerarchitektur und Betriebssystem unabhängig ist.

Die Implementationsbeschreibung der Klasse „Coroutine“ in Abschnitt 3 und insbesondere der Unterabschnitt 3.4 über die Implementierung der Operationen Checkpoint und Rollback setzen ein gründliches Verständnis der C++-Mechanismen *Constructor*, *Destructor*, *Copy-Constructor* und *Assignment-Operator* voraus. Hierzu sei auf Ellis/Stroustrup (1990): *The annotated C++ Reference Manual* (Kapitel 12: Special Member Functions) verwiesen.

In der Einleitung soll zunächst beschrieben werden, was Coroutinen sind und welche Konzepte von Nebenläufigkeit existieren. Dazu muß zunächst der Begriff des *Kontrollflusses in einem Programm* geklärt werden.

### 1.1 Kontrollfluß

Ein wichtiges (vielleicht das wichtigste) Konzept von imperativen Programmiersprachen (z.B. FORTRAN, COBOL, PASCAL, Modula-2, C, C++, SIMULA, ...) ist das Konzept des *Kontrollflusses* (auch *Thread of Control*).

Die Ausführung eines Programmes beginnt stets an einer wohldefinierten Stelle im Programmtext (z.B. in C mit der ersten Anweisung der Funktion `main()`) und die nachfolgenden Anweisungen werden in der Reihenfolge ausgeführt, in der sie im Programmtext notiert sind. Der sogenannte *Programmzähler* (*Program Counter, PC*) zeigt jeweils auf die Stelle (Adresse) im Programmtext, an der die nächste auszuführende Anweisung steht. Das Programm wird beendet,

---

<sup>1</sup>Derzeit existieren Implementierungen für Sun SPARC mit SunOS 4.x & 5.x sowie Intel i386 mit Linux 1.x

wenn keine weiteren Anweisungen vorhanden sind oder eine spezielle Terminierungsanweisung ausgeführt wird.

Es gibt *Kontrollanweisungen*, die diesen linearen Kontrollfluß modifizieren. Die historisch älteste Kontrollanweisung ist der *Sprungbefehl*, der spezifiziert, daß die Ausführung des Programmes an einer anderen Stelle im Programmtext fortgesetzt werden soll. Er bewirkt also, daß der Programmzähler, statt auf die Adresse der nachfolgenden Anweisung zu verweisen, mit einer anderen Adresse geladen wird. Sprungbefehle erlauben bereits die Programmierung von Endlosschleifen. Mit *bedingten Sprungbefehlen* kann der Kontrollfluß in Abhängigkeit vom Programmzustand geändert werden, indem z.B. anhängig vom Wert einer Variablen ein Sprung ausgeführt oder die Ausführung einfach mit der nächsten Anweisung fortgesetzt wird. Erst bedingte Sprungbefehle erlauben die Programmierung von Fallunterscheidungen und terminierenden Schleifen.

Der Kontrollfluß eines Programmes wird im Programmtext durch Kontrollanweisungen beschrieben und während der Ausführung des Programmes durch den Programmzähler repräsentiert:

Der Zustand eines in Ausführung befindlichen Programmes, das zur Modifikation des Kontrollflusses ausschließlich (bedingte) Sprungbefehle verwendet, ist durch die aktuelle Belegung seiner Variablen und den Programmzähler eindeutig bestimmt.

## 1.2 Unterroutinen

Die ausschließliche Benutzung von (bedingten) Sprungbefehlen führt im allgemeinen zu schlecht strukturierten, unübersichtlichen und unverständlichen Programmen, weshalb alle wesentlichen Programmiersprachen Abstraktionen in der Form von *höheren Kontrollanweisungen* (zumindest *while-do-* und *for-*Schleifen) bieten.

Ein deutlich mächtigeres Mittel zur Strukturierung von Programmen und zur Steigerung der Wiederverwendbarkeit von Programmteilen stellen *Unterprogramme* dar. Solche *Unterroutinen* (auch (*Unter-*) *Funktionen* oder *Prozeduren* genannt) werden ausgeführt, wenn sie mit ihrem Namen oder speziellen Anweisungen aufgerufen werden. Die Ausführung einer Unterroutine beginnt stets (analog zur Ausführung des Hauptprogramms) mit der ersten Anweisung des die Unterroutine spezifizierenden Programmtextes, während die Ausführung des aufrufenden Programmteils unterbrochen und erst *nach* Beendigung der Unterroutine mit der dem Aufruf folgenden Anweisung fortgesetzt wird. Dieser Ablauf des Kontrollflusses wird häufig auch bildlich als ein „Herabsteigen des Programmes in die Unterroutine“ beschrieben.

Unterroutinen können selbst wieder andere Unterroutinen (oder sogar sich selbst rekursiv) aufrufen. Zudem ist es im allgemeinen möglich, einer Unterroutine Parameter zu übergeben und bei der Terminierung einer Unterroutine Rückgabewerte von ihr zu übernehmen. Die Parameterübergabe und die Vormerkung der Programmtextstelle, an der nach Terminierung einer Unterroutine die Ausführung fortgesetzt wird (*Rücksprungadresse*), wird typischerweise über den sogenannten *Stack* (Programmstapel) organisiert. Mit jedem Unterprogrammaufruf wird ein neues Objekt auf dem Stack angelegt, welches die Rücksprungadresse und ggf. die Parameter des Aufrufs enthält. Auch lokale Variablen von Unterroutinen werden auf dem Stack verwaltet, sodaß jeder Aufruf einer Unterroutine „seine eigene“ private Menge von Variablen erhält (wichtig für rekursive Aufrufe). Bei Beendigung einer Unterroutine wird die Rücksprungadresse vom zuletzt auf dem Stack angelegten Objekt genommen, dieses Objekt vom Stack entfernt und das Programm an der durch die Rücksprungadresse bezeichneten Stelle fortgesetzt.

Der Kontrollfluß eines Programmes wird nun also im Programmtext durch Kontrollanweisungen

und Unterprogrammaufrufe beschrieben und während der Ausführung des Programmes durch den Programmzähler und den Stack repräsentiert:

Der Zustand eines in Ausführung befindlichen Programmes mit UnterROUTINEN ist durch die aktuelle Belegung seiner Variablen, den Programmzähler und den Inhalt des Stacks eindeutig bestimmt.

### 1.3 Coroutinen

Charakteristisch für die bisher beschriebene Ausführung eines Programms ist der *singuläre, sequentielle* Kontrollfluß: beim Aufruf einer *Unterroutine* wird die Ausführung der aufrufenden Routine (oder des Hauptprogramms) bis zur Beendigung der Unterroutine (einschließlich weiterer dort getätigter UnterROUTINENAUFREFE) unterbrochen.

Der Aufruf einer *Coroutine* bewirkt dagegen, daß sich der Kontrollfluß teilt: die Ausführung der aufrufenden Routine kann sofort fortgesetzt werden und die aufgerufene Coroutine wird unabhängig davon — *nebenläufig* — ausgeführt. Jeder Aufruf einer Coroutine erzeugt also einen neuen Kontrollfluß, für jeden Kontrollfluß existiert ein eigener Programmzähler und ein eigener Stack für die Unterprogrammaufrufe. Man spricht deshalb auch von *multiplem, parallelem* (oder *nebenläufigem*) Kontrollfluß.

Bei Ausführung auf einem RechenSYSTEM mit genügend vielen Prozessoren kann gleichzeitig sowohl die Ausführung der aufrufenden Routine fortgesetzt als auch die Ausführung der aufgerufenen Routine begonnen werden. In diesem Fall spricht man von (*echter*) *Parallelität*.

Auf RechenSYSTEMEN mit einem einzigen Prozessor kann die Parallelität nur durch abwechselnde, stückweise Ausführung der einzelnen Routinen nachgeahmt werden. Dies wird auch als *pseudo-Parallelität* bezeichnet. Hier wird noch feiner danach differenziert, zu welchen Zeitpunkten die Ausführung einzelner Routinen unterbrochen werden kann und wie die nächste auszuführende Routine ausgewählt wird:

- *präemptiv*: eine Routine kann prinzipiell zu jedem Zeitpunkt (z.B. nach jeder Anweisung) „von außen“ unterbrochen werden.
- *non-präemptiv*: die Routine kann nur nach bestimmten Anweisungen unterbrochen werden, die die Routine bestimmt selbst, wann ihr „der Prozessor entzogen werden kann“ und anderen Routinen der Fortschritt ihrer Ausführung erlaubt wird.

Im Kontext von Betriebssystemen spricht man auch von *echtem*, präemptivem Multitasking (z.B. UNIX) und *kooperativem*, non-präemptivem Multitasking (z.B. Microsoft Windows 3.x), Coroutinen werden in Betriebssystemen meist als *Prozesse* oder *Tasks* bezeichnet. Bei Betriebssystemen wird präemptives Multitasking häufig (trotz höherem Implementationsaufwands) non-präemptivem Multitasking vorgezogen, da ersteres die Eigenschaft hat, daß keine einzelne „egoistische“ Routine den Prozessor monopolisieren und alle anderen Routinen vom Fortschritt abhalten kann.

Bei non-präemptivem Multitasking wird weiterhin noch unterschieden, ob die Routine, die den Prozessor aufgibt, explizit einen Nachfolger benennt, oder dieser von einer zentralen Instanz ausgewählt wird: bei Betriebssystemen übernimmt eine spezielle Routine, der sogenannte *Scheduler* oder *Dispatcher* die Auswahl der nächsten fortzusetzenden Routine. Der Scheduler wird automatisch nach jeder Unterbrechung einer „normalen“ Routine aktiviert und plant bei präemptivem

Multitasking mit Hilfe eines Timers auch die nächste Unterbrechung der gewählten Routine. Bei Betriebssystemen kommen zum Coroutinen-Konzept meist noch weitere Konzepte hinzu, beispielsweise getrennte, wechselseitig geschützte Speicherbereiche.

Das Modula-2 und SIMULA zugrundeliegende Coroutinen-Konzept ist non-präemptiv mit expliziter Spezifizierung des Nachfolgers. Beim Aufruf einer Coroutine wird der Aufrufer zunächst *suspendiert* und die Ausführung der aufgerufenen Coroutine begonnen. Die aufgerufenen Coroutine bleibt aktiv, bis sie a) sich selbst suspendiert, wodurch der Aufrufer wieder fortgesetzt wird, oder b) explizit einen Nachfolger benennt, der nun fortgesetzt werden soll (*resume*). In beiden Fällen wird die Ausführung der aufgerufenen Coroutine unterbrochen, bis sie wieder von einer anderen Coroutine zur Fortsetzung benannt wird. Sie setzt dann ihre Ausführung an der letzten Unterbrechungsstelle fort. Dieses Konzept hat sich insbesondere für die Implementierung von Prozeß-orientierten Simulationen (z.B. Class Simulation in SIMULA) bewährt.

Der Zustand eines in Ausführung befindlichen Programmes mit Unter- und Coroutinen ist durch die Zustände aller Coroutinen, also durch die aktuelle Belegung aller Variablen, aller Programmzähler und die Inhalte aller Stacks eindeutig bestimmt.

#### 1.4 Explizite Modifikation von Zuständen und Kontrollflüssen

Alle imperative Programmiersprachen ermöglichen die implizite Modifikation von Kontrollflüssen durch Kontrollanweisungen, meist auch durch Aufrufe von Unterrouتين und manchmal auch durch Coroutinen. Von außen kann ein Kontrollfluß dagegen selten explizit modifiziert, sondern höchstens zeitweilig unterbrochen werden (bei präemptivem Multitasking). Ähnlich ist es mit den Variablen einer Routine: im allgemeinen ist es gerade wünschenswert, daß diese vor Zugriffen von außen geschützt sind und nur von der Routine selbst modifiziert werden können (Kapselung).

Ein Hauptziel dieser Arbeit ist die Untersuchung der Anwendbarkeit von optimistischen Methoden zur parallelen Simulation, die darauf basieren, daß der Zustand von Simulationsprozessen (die durch Coroutinen implementiert sind) gesichert werden kann (*Checkpoint*), die Ausführung des Simulationsprozesses spekulativ fortgesetzt werden kann und ggf. ein zuvor gesicherter Zustand wiederhergestellt werden kann (*Rollback*). Diese Operationen Checkpoint und Rollback sollen insbesondere auch von außen möglich sein.

Da der Zustand einer Coroutine sowohl aus der aktuellen Belegung der Variablen, als auch dem Programmzählern und den Stackinhalten besteht, müssen alle diese Informationen von außen, also durch eine fremde Routine, modifizierbar sein.

SIMULA bietet keine Möglichkeiten zur expliziten Modifikation des Programmzählers und Stacks einer Coroutine. Selbst wenn es gelingen sollte, den Programmzähler und Stack einer Coroutine durch eine andere Coroutine auszulesen und (durch zuvor ausgelesene Information) überschreiben zu können, so ist es doch höchst zweifelhaft, ob damit auch wirklich alle relevanten Informationen berücksichtigt worden sind und ob nicht weitere Informationen in den Tiefen des SIMULA-Laufzeitsystems versteckt sein könnten, die sich den Zugriffen des Programmierers völlig entziehen. Außerdem müssen ja auch alle Attribute und lokalen Variablen der Coroutine gesichert werden.

Dagegen werden solche Manipulationen des Programmzählers und Stacks in einem UNIX-System durch dokumentierte und genormte Systemaufrufe (*setjmp()* zum Sichern des Prozeßkontextes und *longjmp()* zum Wiederherstellen eines gesicherten Prozeßkontextes) unterstützt.

Und bei Verwendung von C oder C++ kann der Programmierer auch sicher sein, daß keine weiteren relevanten Informationen im Laufzeitsystem versteckt sind, da die Mechanismen zur Implementierung von Kontrollflüssen vollständig bekannt sind. Bei der Implementierung des Coroutinen-Konzeptes als abstrakte C++-Klasse läßt sich der Zustand einer Coroutine durch das Kopieren (*Cloning*, realisierbar durch einen sogenannten *Copy-Constructor*) des die Coroutine repräsentierenden Objektes inklusive Stack und Sichern des Prozeßkontextes mittels `setjmp()` realisieren. Durch das *Cloning* werden automatisch auch alle Attribute und lokalen Variablen der Coroutine berücksichtigt.

Leider bieten C und C++ standardmäßig kein Coroutinen-Konzept an, welches dem von SIMULA entspricht (non-präemptiv mit expliziter Nennung des Nachfolgers). Allerdings ließe sich das von UNIX realisierte Coroutinen-Konzept (präemptiv mit wechselseitig geschützten Speicherbereichen) in C und C++ durch die Betriebssystemaufrufe `fork()` und `wait()` nutzen. Das präemptive Multitasking „auszutricksen“, um ein non-präemptives mit expliziter Nennung des Nachfolgers zu realisieren, würde allerdings zu sehr auf Kosten der Leistung gehen, zudem dabei sehr sorgfältig vorgegangen werden müßte, um *Race-Conditions*<sup>2</sup> auszuschließen. Zudem ist die Benutzung von `setjmp()` und `longjmp()` zur Realisierung des gewünschten Coroutinen-Konzeptes vermutlich kaum schwieriger und auch effizienter, da beispielsweise auch auf die Mechanismen zum wechselseitigen Schutz der Speicherbereiche verzichtet werden kann.

De facto wird durch den hier skizzierten Ansatz sogar eine zweistufige Hierarchie von Coroutinen-Konzepten realisiert:

SIMULA-ähnliche Coroutinen laufen innerhalb eines einzigen UNIX-Prozesses, also insbesondere auch innerhalb eines gemeinsamen Speicherbereichs. Innerhalb eines solchen UNIX-Prozesses werden die Coroutinen non-präemptiv und explizit *geschedult*<sup>3</sup> und sequentiell ausgeführt. Zugriffe auf gemeinsame Variablen können somit sehr billig ohne Serialisierung und zusätzlichen Aufwand zur Vermeidung von *Race-Conditions* durchgeführt werden.

Verschiedene UNIX-Prozesse, die eine beliebige Anzahl von SIMULA-ähnlichen Coroutinen enthalten können, besitzen verschiedene wechselseitig geschützte Speicherbereiche und werden präemptiv geschedult. Kommunikation zwischen verschiedenen UNIX-Prozessen (bzw. zwischen Coroutinen, die zu verschiedenen UNIX-Prozessen gehören) kann durch die von UNIX angebotenen (nicht ganz so billigen) IPC-Mechanismen<sup>4</sup> oder (relativ teure) Netzwerkdienste realisiert werden. Dafür haben diese UNIX-Prozesse allerdings das Potential zur echt-parallelen Ausführung.

## 1.5 Leichtgewichtige Prozesse

Das skizzierte zweistufige Konzept mit non-präemptivem Scheduling von innerhalb eines gemeinsamen Speicherbereiches laufenden Coroutinen und präemptivem Scheduling zwischen diesen Gruppen ist im Bereich von Betriebssystemen wohlbekannt und wird unter anderem zur Effizienz- und Effektivitätssteigerung von (File-) Servern in verteilten (Client-Server-) Systemen verwendet.

Die Coroutinen, die innerhalb eines UNIX-Prozesses präemptiv geschedult werden, bezeichnet man dort häufig als *Threads* oder *leichtgewichtige Prozesse*, *Light Weight Processes (LWP)*, um auszudrücken, daß Kontextwechsel und Kommunikation innerhalb eines UNIX-Prozesses billiger

---

<sup>2</sup>Siehe A. S. Tanenbaum (1992): *Modern Operating Systems*, Abschnitt 2.2.1, S. 33–34.

<sup>3</sup>Auswahl des nächsten auszuführenden Prozesses durch den *Scheduler*.

<sup>4</sup>Inter Process Communication

(„leichtgewichtiger“) als zwischen verschiedenen UNIX-Prozessen sind. Ein UNIX-Prozeß, der mehrere solche leichtgewichtige Prozesse enthält wird in dieser Terminologie dann häufig als *Prozeßfamilie* oder auch *Pod*<sup>5</sup> bezeichnet.

Es gibt mehrere (kommerzielle und auch frei verfügbare) LWP-Pakete für UNIX. Manche UNIX-Versionen, z.B. SunOS 5.x (Solaris 2), enthalten mittlerweile bereits standardisierte Betriebssystemaufrufe, die die Erzeugung und Terminierung von, sowie Kommunikation und Scheduling zwischen LWPs unterstützen. Die frei verfügbaren Pakete haben allerdings den Vorteil, daß der Quellcode ebenfalls verfügbar ist und — da die Autoren weitestgehend auf ihr Copyright verzichten — modifizierbar und für ähnliche Anwendungen nutzbar ist.

Diese LWP-Pakete haben sich für typische LWP-Anwendungen, z.B. *Multi-Threaded Server* bewährt. Auch ist mir ein Anwendungsfall außerhalb dieser typischen Betriebssystemdienste bekannt: das frei verfügbare Paket C++SIM (McCue/Little<sup>6</sup>) realisiert diskrete Prozeß-orientierte Simulation durch Imitation von SIMULAs „Class Simulation“ in C++. Es benutzt LWP-Pakete zur Realisierung von SIMULA-ähnlichen Coroutinen.

Leider unterstützt keines dieser Pakete die für optimistische Simulationsprotokolle notwendigen Operationen Checkpoint und Rollback zum Sichern und Zurücksetzen des Zustands eines LWP. Allerdings gab C++SIM Anlaß zur Hoffnung, daß ein auf leichtgewichtigen Prozessen basierendes Coroutinen-Konzept mit den benötigten Erweiterungen realisierbar wäre. Von besonders großer Hilfe war der Quelltext der RexLWP-Library (Stephen Crane<sup>7</sup>), durch dessen Studium und Nutzung einiger Codefragmente es mir gelang, die im folgenden dargestellte LWP-Library zu realisieren.

## 2 Implementierung des LWP-Systems

Die Implementierung des Light Weight Process (LWP) -Systems ist in hohem Maße Rechnerarchitektur- und Betriebssystem-abhängig. Implementierungen wurden für folgende Plattformen vorgenommen:

- Sun SPARC mit Betriebssystem SunOS 4.x
- Sun SPARC mit Betriebssystem SunOS 5.x (Solaris 2)
- PC mit Intel i386 Architektur und Betriebssystem Linux 1.x

Portierungen auf weitere UNIX-Plattformen sind (detaillierte Kenntnis über die entsprechenden Rechnerarchitektur- und Betriebssysteminternia vorausgesetzt) mit relativ geringem Aufwand durchzuführen.

Andererseits verdeckt das LWP-System bereits alle Plattform-spezifischen Details der Implementierung, sodaß die auf dem LWP-System basierende C++-Klasse „Coroutine“ von der konkreten Rechnerarchitektur und Betriebssystem unabhängig implementiert werden konnte.

Eine Übersicht der Funktionen des implementierten LWP-Systems ist in Abb. 1 dargestellt. Außerdem definiert das LWP-System eine Datenstruktur, den sogenannten *Process Control Block (PCB)*, der in Abb. 2 angegeben ist. Der PCB enthält den Prozeßkontext eines LWP,

---

<sup>5</sup> Engl. pod = Herde, Schale, Hülse

<sup>6</sup> Verfügbar im Internet: <ftp://arjuna.ncl.ac.uk/>, <http://ulgham.ncl.ac.uk/C++SIM/homepage.html>

<sup>7</sup> Verfügbar im Internet: <ftp://gummo.doc.ic.ac.uk/rex>

<code>init_lwp</code>	Initialisieren des LWP-Systems, dieser Aufruf muß der erste sein, bevor die übrigen LWP-Funktionen benutzt werden dürfen.
<code>create_lwp</code>	Erzeugung eines neuen LWP.
<code>destroy_lwp</code>	Zerstören eines LWP.
<code>attach_lwp</code>	Aktivieren eines anderen LWP, der Aufrufer wird suspendiert.
<code>save_stack</code>	Kopieren des Stacks.
<code>restore_stack</code>	Zurückschreiben eines kopierten Stacks.

Abbildung 1: Die Funktionen des implementierten LWP-Systems.

Verweise auf den Stack und Beginn des Programmcodes des LWP sowie die Stackgröße (in Bytes).

```

struct pcb
{
    jmp_buf    context;
    void      *stack;
    int       size;
    void      (*entry)();
};

```

Abbildung 2: Process Control Block (PCB)

Der Prozeßkontext wird jedesmal, wenn ein LWP deaktiviert wird (also ein anderer LWP aktiviert wird), in der Datenstruktur `context` seines PCB gespeichert.<sup>8</sup> Sie enthält die aktuellen Belegung aller Prozessorregister inklusive Program Counter (PC) und Stack Pointer (SP). Die Definition des Datentyps `jmp_buf` wird vom UNIX-System importiert<sup>9</sup> und ist extrem architekturabhängig.

Der PCB wird außerhalb des LWP-Systems (also in der Klasse Coroutine) als abstrakter Bezeichner der LWP (sogenannter *handle*) verwendet, es finden außerhalb des LWP-Systems keine Zugriffe auf die Elemente des PCB statt, Referenzen auf PCBs dienen dort also lediglich zur Bezeichnung und Unterscheidung der LWPs. Jeder LWP hat einen eigenen Kontrollfluß, der durch einen PCB eindeutig identifiziert ist.

## 2.1 Initialisierung des LWP-Systems

Ein C- oder C++-Programm beginnt stets mit der Ausführung der Funktion `main()`. Bevor sich der Kontrollfluß das erste mal teilt, also bevor der erste LWP erzeugt wird, existiert im Programm nur dieser einziger Kontrollfluß. Durch den Aufruf der parameterlosen Funktion

<sup>8</sup>Der im PCB des *aktiven* LWP gespeicherte Kontext bezieht sich also auf den Zustand des LWP *vor* der derzeitigen Aktivierung. Ein fataler Fehler wäre es deshalb, wenn ein LWP, beispielsweise um den *aktuellen* Wert *seines* Stackpointers zu ermitteln, auf seinen PCB zugreift.

<sup>9</sup>C-Header-Datei `/usr/include/setjmp.h`

```
struct pcb *init_lwp ()
```

wird das LWP-System initialisiert, indem ein PCB erzeugt und zurückgegeben wird. Dieser PCB identifiziert im weiteren Verlauf den Kontrollfluß, der der Routine `main()` entspricht. Oder anders gesagt: der Prozeß, der `main()` ausführt, wird in einen LWP umgewandelt. Der Stack dieses LWP ist der dem Prozeß ursprünglich vom System zugewiesene Stack.

## 2.2 Erzeugung eines LWP

Ein neuer LWP wird durch den Aufruf der Funktion

```
struct pcb *create_lwp (void (*code)(),
                       int    size,
                       int    argc,
                       char  *argv[],
                       void  *envp)
```

erzeugt. Dieser Aufruf bewirkt, daß ein PCB für den neuen LWP erzeugt und Speicher für einen neuen Stack vom Betriebssystem angefordert wird. Außerdem wird mit initialen Aufrufen von `setjmp()` und `longjmp()` der Stack und der Kontext im PCB initialisiert. Nach dem Aufruf wird die Ausführung des Aufrufers (nicht des Aufgerufenen) fortgesetzt. Die Ausführung des neuen LWP wird also erst begonnen, wenn diese explizit mit `attach_lwp()` veranlaßt wird.

Die Adresse des Startpunkts des LWP wird als erster Parameter `code` übergeben, dies ist im allgemeinen die Adresse einer C-Funktion. Die gewünschte Stackgröße (in Bytes) wird als zweiter Parameter `size` angegeben. Sie wird ggf. aufgerundet. Im PCB wird in jedem Fall die tatsächliche Größe des Stacks vermerkt.

Es ist möglich, einem LWP bei der Erzeugung Parameter und einen Zeiger auf seine Umgebung zu übergeben. Dies geschieht in genau derselben Weise, wie die Parameterübergabe an die Hauptfunktion `main()` eines C-Programmes, falls diese wie üblich durch

```
main (int argc, char *argv[])
```

deklariert ist. Dadurch ist es auf sehr einfache Weise möglich, C-Hauptfunktionen, auch wenn sie mit Parametern aufgerufen werden und auf ihre Umgebung zugreifen, in Funktionen umzuwandeln die als LWP gestartet werden können: lediglich der Name `main` muß abgeändert werden.

Bei der Implementierung der Klasse „Coroutine“ wird dieser Mechanismus „mißbraucht“, um den neu erzeugten LWP mit einer C++-Klassenmethode starten zu lassen:

Jede Methode einer C++-Klasse besitzt implizit als ersten Parameter einen Verweis auf das Objekt, an dem diese Methode aufgerufen wird, den `this`-Zeiger. Diese Parameterübergabe ist für einen C++-Programmierer unsichtbar. Eine C++-Methode *ohne* Parameter besitzt also stets implizit den `this`-Zeiger als unsichtbaren Parameter. Wenn nun eine solche C++-Methode als Startpunkt eines LWP angegeben werden soll, dann muß dafür Sorge getragen werden, daß der `this`-Zeiger dem LWP bei der ersten Aktivierung zur Verfügung steht. Dies kann durch „Mißbrauch“ des `argc`-Parameter erreicht werden:

```
create_lwp ((void*)(void))(& anObj->aMethod), size, (int)anObj, 0, 0)
```

## 2.3 Zerstörung eines LWP

Ein LWP wird automatisch zerstört, wenn die Funktion, deren Startadresse beim Erzeugen des LWP angegeben wurde, terminiert. In diesem Fall wird automatisch wieder der initiale LWP der Funktion `main()`, dessen PCB mit dem Aufruf von `init_lwp()` erzeugt wurde, aktiviert. Durch den Aufruf von

```
void destroy_lwp (struct pcb *)
```

kann ein LWP auch zerstört werden, wenn er noch nicht terminiert ist. Als Parameter muß der den LWP kennzeichnende PCB angegeben werden. Bei automatischer und expliziter Zerstörung wird stets der vom Kontext und Stack des LWP belegte Speicher wieder freigegeben.

## 2.4 Aktivierung eines LWP

Ein LWP wird aktiviert, indem die Funktion

```
void attach_lwp (struct pcb *)
```

mit dem PCB des zu aktivierenden LWP als Parameter aufgerufen wird. Dies führt dazu, daß zunächst der Kontext des derzeit aktiven (die Funktion `attach_lwp()` aufrufenden) LWP mit einem Aufruf von `setjmp()` in dessen PCB gesichert wird und dann der im PCB des zu aktivierenden LWP gespeicherte Kontext mit einem Aufruf von `longjmp()` wiederhergestellt wird.

## 2.5 Kopieren von Stacks

Für die „normale“ Verwendung von LWPs ohne Checkpoint und Rollback ist es ausreichend, bei jeder Aktivierung eines LWP den *Kontext* des aktiven LWP zu sichern und den gesicherten Kontext des zu aktivierenden LWP wiederherzustellen (*Kontextwechsel*). Es müssen keine Stackinhalte kopiert werden, da jeder LWP einen eigenen Speicherbereich für seinen Stack besitzt und beim Kopieren des Kontextes automatisch der Stackpointer (ein Register im Prozessor) kopiert wird.

Erst für die Implementierung von Checkpoint und Rollback in der Klasse „Coroutine“ wird ein Umkopieren von Stackinhalten notwendig, um diese sichern und ggf. wiederherstellen zu können, da eine Coroutine während spekulativer Ausführung den Stack (durch Aufrufe oder Terminierungen von Unterroutinen) modifizieren kann und bei einem Rollback diese Effekte wieder rückgängig gemacht werden müssen.

Die Größe der Stacks von LWPs wird bei der Erzeugung der LWPs manuell angegeben. Dies ist sinnvoll, da der Stack für einen LWP, der eine tiefe Aufrufhierarchie hat (z.B. rekursive Aufrufe, manche Betriebssystemaufrufe), deutlich größer sein muß als ein Stack für einen LWP mit flacher Aufrufhierarchie. Da Parameter und lokale Variablen bei Aufrufen von Unterroutinen ebenfalls auf dem Stack organisiert werden, ist der Speicherbedarf der Stacks von verschiedenen LWPs in hohem Maße variabel. Typischerweise werden Größen von 10 bis 100 KBytes benötigt.

Bekannterweise führt ein zu klein dimensionierter Stack zu einem Überlauf (*Stackoverflow*), der im allgemeinen einen Programmabbruch zur Folge hat. Andererseits ist es auch nicht ratsam, mit der (meist knappen) Resource „Hauptspeicher“ zu verschwenderisch umzugehen, insbesondere wenn geplant ist, mit hunderten von LWPs gleichzeitig im System zu arbeiten. Das Problem wird vollends kritisch, wenn berücksichtigt wird, daß von jeder einzelnen Coroutine möglicherweise

mehrere (10? 100?) Zustandskopien gleichzeitig verwaltet werden müssen. Deshalb sollte die Stackgröße, die bei der Erzeugung der LWPs manuell angegeben wird, *sorgfältig* gewählt werden.

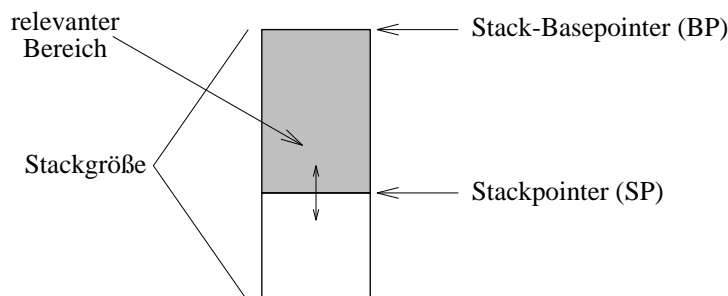


Abbildung 3: Ein Stack, der (wie unter SunOS und Linux üblich) von oben nach unten wächst.

Beim Kopieren eines Stacks genügt es, nur den *relevanten* Teil des Stacks zu kopieren. Dieser Bereich umfaßt die zwischen dem *Stackpointer* und *Stack-Basepointer* (dies sind zwei Prozessorregister) liegenden Speicherstellen (Abb. 3). In ihm liegen die Informationen, die die aktuelle Aufrufhierarchie erzeugt hat. Ein weiterer Unterroutinenaufruf würde den Stack wachsen lassen, indem sich der Stackpointer weiter vom Stack-Basepointer entfernt, eine Terminierung eines Aufrufs läßt den Stackpointer wieder näher an den Stack-Basepointer rücken. Außerhalb dieses Bereichs vom Stackpointer bis zum Stack-Basepointer liegende Informationen stammen also von früheren, bereits terminierten Aufrufen und sind irrelevant.

Eine weitere Möglichkeit zur noch sparsameren Nutzung des Speichers würde ein *inkrementelles* Kopieren der Stackinhalte darstellen. Da dieses jedoch insbesondere bzgl. des Wiederherstellen des Stackinhalts und Löschen von nicht länger benötigten Kopien von recht großer Komplexität ist, wurde es zunächst nicht weiter betrachtet.<sup>10</sup>

Die Funktion

```
void save_stack (struct pcb *backup, struct pcb *orig)
```

kopiert den relevanten Bereich des Stacks des durch den PCB *orig* bezeichneten LWPs in einen neuen PCB *backup*, der vor dem Aufruf angelegt worden sein muß. Während im „echten“ PCB *orig* des LWP die tatsächliche Stackgröße vermerkt ist, wird in dem PCB *backup* die Größe des kopierten Bereichs eingetragen. Dies ermöglicht der Funktion

```
void restore_stack (struct pcb *orig, struct pcb *backup)
```

den kopierten Stackbereich aus dem PCB *backup* wieder in den Stack des durch den PCB *orig* bezeichneten LWPs zurückzukopieren.

Da beide Funktionen ebenfalls extrem architekturabhängig sind, wurden sie in das LWP-Paket aufgenommen.

---

<sup>10</sup>Es wäre ein sehr aufwendiges Versionsmanagement notwendig: die einzelnen Kopien müßten verkettet werden und beim Löschen einer Kopie müßte die folgende Kopie um den Inhalt der zu löschenden Kopie, der ja nicht in den nachfolgenden Kopien enthalten ist, ergänzt werden.

### 3 Implementierung der Klasse Coroutine

Nun wird die C++-Klasse „Coroutine“ und deren Implementierung mit Hilfe des LWP-Systems beschrieben. Im Gegensatz zu anderen Implementierungen läßt sich der Zustand einer solchen Coroutine sichern (*Checkpoint*) und später wiederherstellen (*Rollback*), während die Coroutine zwischenzeitlich *spekulativ* fortgesetzt worden sein kann.

Das LWP-System verdeckt bereits alle Plattform-spezifischen Details der Implementierung, so daß die Implementierung der C++-Klasse „Coroutine“ von der konkreten Rechnerarchitektur und vom Betriebssystem unabhängig ist.

```
class coroutine
{
    private:
        struct pcb    *tid;
        int           co_state;
        void          start      ();
        virtual void  body      () {};
        void          (*lastrites) (coroutine*);

    public:
        coroutine*   current    ();
        int          state      ();
        friend void  resume     (coroutine*);
        void         suspend    ();

        virtual     coroutine   ();
        virtual     ~coroutine   ();

        coroutine& operator=   (const coroutine&);

        virtual    coroutine*   checkpoint  ();
        virtual    void         rollback   (coroutine*);
};
```

Abbildung 4: Die Definition der Klasse „Coroutine“

Die Definition der Klasse „Coroutine“ ist in Abb. 4 dargestellt. Sie wird benutzt, indem weitere Klassen von ihr abgeleitet (spezialisiert) werden und diese die virtuelle Funktion `body()` redefinieren. Oder anders gesagt: Klassen können das Konzept „Coroutine“ nutzen, indem sie diese Klasse als (eine) Vaterklasse angeben. Die Funktion `body()` ist das Analogon zur Hauptfunktion `main()` gewöhnlicher *single threaded C-* oder *C++-Programme* und enthält den Rumpf der Coroutine.

### 3.1 Beispiel: Produzenten–Konsumenten–Szenario

Bevor diese Klassendefinition im Detail erläutert und die Implementierung beschrieben wird, soll ein einfaches Beispiel die Benutzung dieser Klasse verdeutlichen. Als motivierendes Beispiel sei ein stark vereinfachtes *Produzenten–Konsumenten–Szenario* betrachtet (Abb. 5).

#### 3.1.1 Beispiel 1

Der Konsument verarbeitet einen vom Produzenten erzeugten und in einem Puffer (`int item`) abgelegten Gegenstand und aktiviert anschließend den Produzenten, wodurch er selbst bis zu seiner nächsten Aktivierung passiviert wird.

Der Produzent erzeugt jeweils einen Gegenstand (hier die Zweierpotenzen von  $2^1$  bis  $2^n$ ), den er im Puffer ablegt und aktiviert dann den Konsumenten, wodurch er selbst bis zu seiner nächsten Aktivierung passiviert wird.

Die Klasse `producer` hat einen selbstdefinierten Konstruktor, der als Parameter die Anzahl erzeugender Gegenstände hat. Wenn der Produzent die angegebene Anzahl von Gegenständen erzeugt hat, terminiert die `for`-Schleife und somit die Funktion `body()`. Diese Terminierung einer Coroutine führt automatisch zur Reaktivierung der Hauptfunktion `main()`, die dann auch sofort terminiert wird, da keine weiteren Anweisungen folgen (siehe unten).

Das Programm erzeugt schließlich (vorausgesetzt es wurde erfolgreich kompiliert und gestartet<sup>11</sup>) die folgende Ausgabe:

```
produced 1 items
consumed item 2
produced 2 items
consumed item 4
produced 3 items
consumed item 8
produced 4 items
consumed item 16
produced 5 items
consumed item 32
```

---

<sup>11</sup>Falls die entsprechenden Include- und bereits kompilierten Object-Files im Unterverzeichnis `lib` zu finden sind: `gcc -I lib -o example example.cc lib/lwp.o lib/coroutine.o`

```

#include <stdio.h>
#include "coroutine.h"

int      item = 1;
coroutine *the_producer;
coroutine *the_consumer;

class consumer : public coroutine
{
private:
    void body() // Activities of all Consumers
    {
        while (1)
        {
            printf("consumed item %d\n", item);
            resume(the_producer);
        }
    }
};

class producer : public coroutine
{
private:
    int n, i; // Private Attributes of a Producer

    void body() // Activities of all Producers
    {
        for (i = 1; i <= n; i++)
        {
            item = 2 * item;
            printf("produced %d items\n", i);
            resume(the_consumer);
        }
    }

public:
    producer (int no_items) { n = no_items; } // Constructor initializes n
};

main()
{
    the_consumer = new consumer();
    the_producer = new producer(5);
    resume (the_producer);
}

```

Abbildung 5: Produzenten–Konsumenten–Beispiel

### 3.1.2 Beispiel 2

Nun wird das Beispiel um Aufrufe der Operationen `checkpoint()` und `rollback()` ergänzt. Abbildung 6 zeigt die an der Klassendefinition `consumer` vorgenommenen Ergänzungen:

```
class consumer : public coroutine
{
    private:
        coroutine* aBackup;

    void body()
    {
        while (1)
        {
            printf("consumed item %d\n", item);
            if (item == 4) {
                printf("Now checkpointing the producer\n");
                aBackup = the_producer->checkpoint();
            }
            if (item == 8) {
                printf("Now rolling back the producer\n");
                the_producer->rollback(aBackup);
            }
            resume(the_producer);
        }
    }
};
```

Abbildung 6: Ergänzungen in der Klasse `consumer`

Der Konsument soll den Zustand des Produzenten sichern, sobald dieser das `item == 4` produziert hat und den gesicherten Zustand wiederherstellen, nachdem der Konsument das `item == 8` produziert hat, zwischenzeitlich also fortgesetzt worden ist.

Das Programm erzeugt allerdings noch *nicht* die gewünschte Ausgabe:

```
produced 1 items
consumed item 2
produced 2 items
consumed item 4
Now checkpointing the producer
produced 3 items
consumed item 8
produced 4 items
consumed item 16
Now rolling back the producer
produced 5 items
consumed item 32
```

### 3.1.3 Beispiel 3

Offensichtlich bleibt der Produzent im 2. Beispiel von den Operationen `checkpoint()` und `rollback()` unbeeindruckt. Das liegt daran, daß die Methoden `checkpoint()` und `rollback()`, die aufgerufen werden, die in der Klasse `coroutine` definierten Methoden sind, die natürlich die lokalen Attribute `int n, i` der Klasse `producer` nicht kennen.

```
class producer : public coroutine
{
    private:
        int n, i;

        void body()
        {
            for (i = 1; i <= n; i++)
            {
                item = 2 * item;
                printf("produced %d items\n", i);
                resume(the_consumer);
            }
        }

    public:
        producer (int no_items) { n = no_items; } // Constructor
        virtual coroutine* checkpoint () { return new producer (*this); };
        virtual void rollback (coroutine* c) { *this = *((producer*) c); };
};
```

Abbildung 7: Ergänzungen in der Klasse `producer`

Damit die lokalen Attribute `int n, i` der Klasse `producer` mitgesichert werden, müssen in der Klasse `producer` die virtuellen Deklarationen der Methoden `checkpoint()` und `rollback()` wiederholt werden. Abbildung 7 zeigt diese an der Klassendefinition `producer` vorgenommenen Ergänzungen. Das Programm erzeugt nun die folgende Ausgabe:

```
produced 1 items
consumed item 2
produced 2 items
consumed item 4
Now checkpointing the producer
produced 3 items
consumed item 8
produced 4 items
consumed item 16
Now rolling back the producer
produced 3 items
consumed item 32
produced 4 items
consumed item 64
produced 5 items
consumed item 128
```

### 3.2 Konstruktor und Destruktor

Der Konstruktor `coroutine()` erzeugt einen neuen leichtgewichtigen Prozess (LWP), indem er die LWP-Funktion `create_lwp()` mit der nicht-virtuellen Funktion `start()` als Startadresse aufruft. Die von `create_lwp` zurückgegebene Referenz auf den Process Control Block (PCB) wird in dem privaten Attribut `tid` gespeichert.

Außerdem sorgt der Konstruktor ggf. zuvor durch den einmaligen Aufruf von `init_lwp()` dafür, daß das gesamte LWP-System initialisiert wird, falls dies nicht schon vorher (durch einen anderen Aufruf des Konstruktors) geschehen ist. Es ist wichtig, sich der Tatsache bewußt zu sein, daß für die Hauptfunktion `main()` dadurch zwar ein LWP erzeugt wird, `main()` aber selbstverständlich *nicht* in eine Coroutine (ein Objekt von der Klasse „Coroutine“) umgewandelt werden kann.

Die neu erzeugte Coroutine wird nicht unmittelbar aktiviert, was ihr Zustand `co_state = CO_BORN`, der über die Funktion `state()` abfragbar ist, ausdrückt. Bei der ersten Aktivierung (durch `resume()`, siehe unten) beginnt die Ausführung der Funktion `start()`, die einfach den Zustand der Coroutine in `co_state = CO_ALIVE` ändert und die virtuelle Funktion `body()` aufruft (Abb. 8).

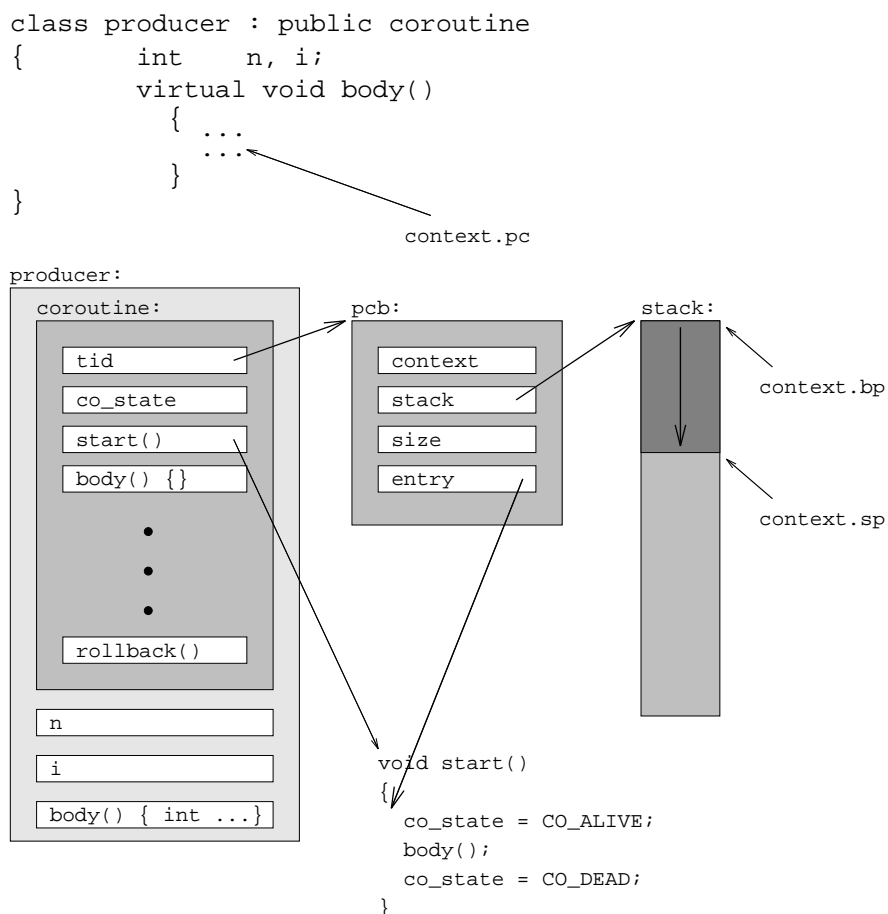


Abbildung 8: Repräsentation der Coroutine `producer`.

Dies ist die Funktion `body()`, wie sie in der *abgeleiteten* Klasse definiert ist!

Wenn diese Funktion `body()` terminiert ist, so wird dies durch den Zustand `co_state = CO_DEAD` gekennzeichnet. Die Coroutine wird nun *nicht* automatisch zerstört, da sie ja durch ein Rollback „reinkarniert“ werden könnte. Stattdessen wird unmittelbar nach Terminierung von `body()` die Funktion `lastrites()` (falls sie definiert ist) ausgeführt<sup>12</sup>. In dieser Funktion können sogenannte *Pseudo-Terminalisierungen* vorgenommen werden. Die Funktion `lastrites()` wurde nicht als virtuelle Funktion, sondern als Referenz auf eine Funktion in die Klasse integriert, um die Möglichkeit offen zu halten, die Pseudo-Terminalisierungen dynamisch ändern zu können.

Erst wenn die Coroutine explizit durch Aufruf des Destruktors `~coroutine()` gelöscht wird, wird auch der ihr zugeordnete LWP durch einen Aufruf von `destroy_lwp()` zerstört und somit der belegte Speicherplatz freigegeben. Nun ist die Coroutine unwiederbringlich zerstört und kann auch durch ein Rollback nicht mehr reinkarniert werden.

Für abgeleitete Klassen können Konstruktoren und Destruktoren selbstdefiniert werden, oder sie werden vom C++-Compiler automatisch erzeugt. Per Definitionem rufen Konstruktoren (Destruktoren) von abgeleiteten Klassen stets die Konstruktoren (Destruktoren) der Vaterklasse auf, bevor (nachdem) die Anweisungen im Rumpf des Konstruktors (Destruktors) ausgeführt werden (wurden). Auf diese Weise vererben sich die beschriebenen Mechanismen des Konstruktors und Destruktors der Klasse „Coroutine“ automatisch auf abgeleitete Klassen.

Es sei noch deutlich darauf hingewiesen, daß Terminalisierungen, die in den Destruktoren von abgeleiteten Klassen spezifiziert sind, im Gegensatz zu den Pseudo-Terminalisierungen `lastrites()` stets höchstens einmal ausgeführt werden.

### 3.3 Aktivierung und Passivierung

Eine neu erzeugte Coroutine wird nicht automatisch aktiviert, sondern erst durch einen Aufruf der Funktion `resume()`, die eine Referenz auf die zu aktivierende Coroutine als Parameter erhält.

Die Funktion `resume()` kann *nicht* als Klassen-Methode implementiert werden, da sonst eine das `resume()` aufrufende Coroutine *A* im Code der zu aktivierende Coroutine *B* passiviert würde, was, falls *B* nun endgültig terminiert und zerstört würde, bei der nächsten Aktivierung der im nun obsoleten Code stehen Coroutine *A* zu schweren Fehlern führen könnte.

Wird versucht, eine Coroutine mit `resume()` zu aktivieren, deren `body()` bereits terminiert ist (Zustand `co_state = CO_DEAD`), so wird das Programm mit einer Fehlermeldung abgebrochen.

Durch einen Aufruf von `suspend()` ohne Parameter wird die aktive Coroutine passiviert und der LWP, der zur Hauptfunktion `main()` gehört, aktiviert.

Die Identität der derzeit aktiven Coroutine kann mit der Funktion `current()` ermittelt werden. Sie gibt eine Referenz (den `this`-Zeiger) auf das Objekt von der Klasse „Coroutine“, das gerade aktiv ist, zurück, bzw. `NULL`, falls *keine* Coroutine, sondern der LWP der Hauptfunktion `main()` aktiv ist. Diese Funktion ist nützlich, wenn Coroutinen Methoden von anderen Coroutinen aufrufen können: der `this`-Zeiger bezeichnet dann die Coroutine, deren Methode aufgerufen wird und der von `current()` zurückgegebene Zeiger bezeichnet die Coroutine, in deren LWP dieser Aufruf gerade stattfindet.

---

<sup>12</sup>engl. *last rites* = Sterbesakramente

### 3.4 Checkpoint und Rollback

Checkpoint und Rollback werden durch einen sogenannten *Copy-Constructor* bzw. *Assignment-Operator* implementiert:

```
coroutine (const coroutine&);
coroutine& operator= (const coroutine&);
```

Ein Copy-Constructor dient dazu, von einem Objekt eine Kopie zu erzeugen (*Cloning*). Durch einen Assignment-Operator werden Zuweisungen zwischen Objekten der gleichen Klasse definiert. Copy-Constructoren und Assignment-Operatoren können sowohl selbstdefiniert als auch automatisch vom C++-Compiler erzeugt werden. Beide Möglichkeiten werden bei der Implementierung der Checkpoint-Operation und Rollback-Operation ausgenutzt.

#### 3.4.1 Checkpoint durch einen selbstdefinierten Copy-Constructor

Der Copy-Constructor wird mit der Coroutine, deren Zustand gesichert werden soll, als Parameter aufgerufen und gibt ein neues Objekt der Klasse „Coroutine“ zurück, welches alle relevanten Zustands-Informationen des LWPs und des Coroutinen-Objektes enthält und als *Backup-Coroutine* bezeichnet wird:

```
aBackup = new coroutine (aCoroutine);
```

In der Implementierung des Copy-Constructors (Abb. 9) bezeichnet der `this`-Zeiger die Kopie des Objektes `orig`, die automatisch erzeugt wird, bevor mit der Ausführung der Anweisungen im Rumpf des Copy-Constructors begonnen wird.

```
coroutine::coroutine(const coroutine& orig)
{
    this->co_state      = orig.co_state;
    this->tid           = new pcb;
    this->tid->entry    = NULL;
    this->tid->context  = orig.tid->context;
    save_stack(this->tid, orig.tid);
}
```

Abbildung 9: Die Implementierung des Copy-Constructors

Dort wird zunächst der `co_state` (`CO_BORN`, `CO_ALIFE`, `CO_DEAD`) gesichert, indem er von der zu sichernden Coroutine in das kopierte Objekt geschrieben wird. Dann wird ein neuer Process Control Block (PCB) erzeugt, der den LWP-Kontext und gesicherten LWP-Stack aufnimmt.

Es wird *kein* neuer LWP für die Kopie erzeugt!

Die Kopie der „echten“ Coroutine wird als solche durch `this->tid->entry = NULL` gekennzeichnet und darf nur als abstraktes Objekt, welches eine Zustandssicherung der „echten“ Coroutine speichert und der Rollback-Operation als Parameter übergeben werden kann, verwendet werden.

#### 3.4.2 Rollback durch einen selbstdefinierten Assignment-Operator

Der Assignment-Operator zum Zurücksetzen einer Coroutine wird benutzt, indem als rechte Seite der Zuweisung die Coroutinen-Kopie, die den gesicherten Zustand der „echten“ Coroutine

enthält, angegeben wird. Als linke Seite der Zuweisung muß die zurückzusetzende Coroutine angegeben werden:

```
aCoroutine = aBackup;
```

Als rechte Seite *darf nur* eine mit dem Copy-Constructor erzeugte Backup-Coroutine benutzt werden!

In der Implementierung des Assignment-Operators (Abb. 10) wird das Objekt der Klasse „Coroutine“, das auf der linken Seite der Zuweisung angegeben ist, mit dem `this`-Zeiger bezeichnet. Die Kopie, die die gesicherten Zustandsinformationen enthält, heißt `theCopy`.

```
coroutine& coroutine::operator=(const coroutine& theCopy)
{
    if (theCopy.tid->entry != NULL) raise_error_exception();
    this->tid->context = theCopy.tid->context;
    restore_stack(this->tid, theCopy.tid);
    this->co_state    = theCopy.co_state;
    return *this;
}
```

Abbildung 10: Die Implementierung des Assignment-Operators

Zunächst wird sichergestellt, daß es sich bei dem Objekt auf der rechten Seite der Zuweisung tatsächlich um eine Backup-Coroutine — und nicht etwa eine „echte“ Coroutine — handelt. Dann wird der gesicherte LWP-Kontext und LWP-Stack der Coroutine, die auf der linken Seite angegeben ist, wiederhergestellt und schließlich der `co_state` (`CO_BORN`, `CO_ALIFE`, `CO_DEAD`) restauriert.

### 3.4.3 Fortsetzung auf abgeleitete Klassen

Copy-Constructoren und Assignment-Operatoren werden *automatisch* vom C++-Compiler auf abgeleitete Klassen fortgesetzt, siehe Ellis/Stroustrup (1990): *The annotated C++ Reference Manual* (Abschnitt 12.8, S. 295):

Für eine abgeleitete Klasse wird automatisch ein Copy-Constructor definiert, wenn

1. kein selbstdefinierter Copy-Constructor vorhanden ist,
2. alle Vaterklassen<sup>13</sup> einen Copy-Constructor besitzen und
3. für die Klassen alle Unterobjekte (*Part-Objects*, nicht Referenzen) Copy-Constructoren existieren.

Der automatisch definierte Copy-Constructor ruft die Copy-Constructoren aller Vaterklassen und Klassen der Unterobjekte auf. Analoges gilt für Assignment-Operatoren.

Auf diese Weise wird erreicht, daß beim Zustandssichern und Wiederherstellen eines Objektes einer von `coroutine` abgeleiteten Klasse automatisch auch alle in dieser abgeleiteten Klasse zusätzlich definierten Attribute berücksichtigt werden.

---

<sup>13</sup>C++ kennt *Multiple Inheritance* = Mehrfachvererbung

Die Methoden `checkpoint()` und `rollback()` dienen nur der Notationsvereinfachung, um Fehler bei Aufrufen des Copy-Constructors und Assignment-Operators — insbesondere bei der Verwendung von Objekt-Referenzen statt Objekt-Instanzen — zu vermeiden:

```
aBackupRef = aCoroutineRef->checkpoint();
aCoroutineRef->rollback(aBackupRef);
```

ist äquivalent zu

```
*aBackupRef = new coroutine(*aCoroutineRef);
*aCoroutineRef = *aBackupRef;
```

`checkpoint()` und `rollback()` werden leider nicht automatisch vom C++-Compiler auf abgeleitete Klassen fortgesetzt und müssen manuell folgendermaßen (hier für eine abgeleitete Klasse mit Namen `XX`) definiert werden:

```
virtual coroutine* XX::checkpoint ()           { return new XX(*this); };
virtual void       XX::rollback (coroutine* c) { *this = *((XX*)c);   };
```

Wird `checkpoint()` oder `rollback()` an einem Objekt einer abgeleiteten Klasse aufgerufen, für die diese Definitionen nicht manuell vorgenommen worden sind, so wird lediglich die `checkpoint()` bzw. `rollback()`-Operation der Vaterklasse aufgerufen, wodurch die in der abgeleiteten Klasse zusätzlich definierten Attribute nicht berücksichtigt werden.

## 4 Zusammenfassung

In dieses Kapitel wurde die Implementierung eines SIMULA-ähnlichen Coroutinen-Konzeptes als C++-Klasse mit Hilfe von *leichtgewichtigen Prozessen* beschrieben. Im Gegensatz zu anderen Implementierungen läßt sich der Zustand einer solchen Coroutine sichern und später wiederherstellen, während die Coroutine zwischenzeitlich fortgesetzt worden sein kann.

Die Implementierung unterteilt sich in zwei Abschnitte: zunächst wurde ein LWP-System realisiert, dessen Implementierung in hohem Maße Plattform-abhängig ist. Es verdeckt allerdings bereits solche Details der Implementierung, sodaß die auf dem LWP-System basierende Implementierung der C++-Klasse „Coroutine“ von der konkreten Rechnerarchitektur und Betriebssystem unabhängig ist.