Technische Universität Chemnitz-Zwickau

Fakultät für Informatik

TR-95-07

Nichtprozedurale Ablaufformen in imperativen Sprachen

– Coroutinen und preemptive Threads in C –

Sven Graupner

August 1995

Chemnitzer Informatik-Berichte

Kurzfassung

In imperativen Programmiersprachen sind Prozeduren Grundeinheiten der funktionalen Struktur von Programmen und fixieren gleichzeitig die sequentielle Ablaufform des späteren Systems auf der Maschine.
In diesem Beitrag wird am Beispiel der Sprache C ein Weg gezeigt, mit einer unveränderten Sprache nichtprozedurale Ablaufformen in einem Anwendungssystem herzustellen, indem die Ablaufumgebung von Prozeduren manipuliert wird. Es werden drei Ablaufformen betrachtet: *Coroutinen*, *entkoppelte Coroutinen* und *Leichtgewichtsprozesse (preemptive Threads)*. Insbesondere die zuletzt genannte Ablaufform gewinnt für parallele und verteilte Systeme immer mehr an Bedeutung. Für Unix existieren bereits zahlreiche Thread-Bibliotheken, deren innere Wirkprinzipien jedoch verborgen sind. Das Ziel dieses Beitrages ist die vollständige Offenlegung dieser Prinzipien, um deren Verständnis zu fördern.
Nach der Analyse der Ablaufumgebung von Prozeduren werden schrittweise Manipulationen vorgestellt, die aufeinander aufsetzend die Ablaufformen für Coroutinen, entkoppelte Coroutinen und preemptive Threads ergeben. Es werden ausschließlich bekannte Grundelemente der Sprache C und der Standard-C-Bibliothek benutzt und alle Details offengelegt.

Schlüsselwörter: Sprache C, Ablaufumgebung, Prozeduren, Coroutinen, entkoppelte Coroutinen, Prozesse, Leichtgewichtsprozesse, Scheduling, Entzug, Wettlaufbedingungen, wiedereintrittsfähige Funktionen

keywords: C language, execution environment, procedures, coroutines, decoupled coroutines, processes, light weight processes, threads, scheduling, preemption, race conditions, reentrant functions

Bibliographische Angaben: 51 Seiten, 22 Abbildungen, 30 Literaturverweise.

Dipl. Inf. Sven Graupner

Technische Universität Chemnitz-Zwickau
Fakultät für Informatik
Lehrstuhl Betriebssysteme
09107 Chemnitz
email: sgr@informatik.tu-chemnitz.de

Inhaltsverzeichnis

1. Einführung

In Betriebssystemen wird die Abarbeitung der durch das Programm vorgegebenen Folge von Instruktionen als *"Prozeß"* verstanden. Das zeitliche Voranschreiten der Abarbeitung ist somit durch *"Aktivität"* gekennzeichnet, d.h., ein aktives Element (Prozessor) führt eine Folge von Instruktionen (Operationen), die im Programm vorgegeben sind, über Daten aus (Zustandsänderung). Zu einem Zeitpunkt wird in einem Prozeß genau eine Instruktion abgearbeitet. Für die Existenz eines Prozesses sind (mind.) die zwei Betriebsmittel "Speicher" und "Prozessor" (in verschiedenen Ausprägungen) erforderlich. Instruktionen befinden sich wie Daten im Speicher. Die Adresse der gerade abgearbeiteten Instruktion wird durch den Wert des Befehlszählers des Prozessors angezeigt. Nach der Abarbeitung wird der Befehlszähler inkrementiert (Sequenz) oder ein neuer Wert durch eine Instruktion gesetzt (Sprung), um die folgende Instruktion im Speicher zu adressieren. Dieser Ablauf kennzeichnet den *"Steuerfluß"* auszuführender Instruktionen (*thread of control*). Alle drei Begriffe "Prozeß", "Aktivität" oder "Steuerfluß" stehen als Synonym für den geschilderten Ablauf der Abarbeitung von Instruktionen durch einen Prozessor (Prozessor = *"Aktivitätsträger"*).

In einem Rechnersystem, auch in einem typischen Ein-CPU-System, existieren zu einem Zeitpunkt mehrere Prozesse, Aktivitäten oder Steuerflüsse, wenn man *alle* internen und externen Aktivitätsträger des Systems betrachtet, die zu Zustandsänderungen im System führen. Dazu zählen die CPU(s) des Rechnersystems, aber auch Coprozessoren, Controller, externe Geräte als "reale" Aktivitätsträger des Systems, auf denen Prozesse zeitlich *"echt parallel"* zueinander ablaufen und zwischen denen Wechselwirkungen bestehen. Es ist die Aufgabe des Betriebssystems, diese Prozesse zu steuern und höheren Ebenen des Systems weitgehend zu verbergen.

Auf einem Prozessor können sich auch mehrere Prozesse, zeitlich versetzt, in Abarbeitung befinden. Dieses Verfahren wurde ursprünglich eingeführt, um Wartezeiten eines Prozesses bei Ein- bzw. Ausgaben von Daten durch Weiterbearbeitung eines anderen Programmes zu nutzen (multi-programming).
Entkoppelt man die zur Abarbeitung anstehenden Prozesse soweit, daß in den abgearbeiteten Programmen keinerlei Berücksichtigung der anderen Prozesse (bzgl. Speicher, Prozessor und anderer Betriebsmittel) mehr sichtbar ist, kann man in dieser Betrachtungsebene von *virtuellen Prozessoren* sprechen, die jeweils ein Programm abarbeiten. Es muß eine Abbildung der virtuellen auf den (die) realen Prozessor(en) erfolgen.
Durch zeitlich bzw. räumlich geteilt genutzte reale Betriebsmittel können diese (virtuell) vervielfacht und jedem Prozeß exklusiv zur Verfügung gestellt werden. Ein Prozeß wird von einem virtuellen Prozessor abgearbeitet, besitzt virtuellen Speicher und arbeitet mit virtuellen Geräten. Bezüglich des Prozessors spricht man bei zeitlich versetzter Abarbeitung von *nebenläufigen (concurrent)* Prozessen. Es existieren zu einem Zeitpunkt mehrere, noch nicht beendete Prozesse, die zeitlich versetzt voranschreiten.

Sequentielle Programmiersprachen und Programmiersysteme stellen oft die Abstraktion eines einzigen Steuerflusses und damit eines einzigen Prozesses bereit. Dies ist eine sinnvolle Vereinfachung, solange keine Abhängigkeiten zu anderen Prozessen des Systems bestehen, die eine Koordination mit diesen erfordern. Ist das jedoch notwendig, können die entsprechenden Koordinationsmechanismen oft nur ungenügend in Programmen abgebildet werden, da entsprechende sprachliche Ausdrucksmittel fehlen.

Auf der Ebene der Anwendungsprogramme findet man jedoch immer häufiger mehrere zu koordinierende Prozesse vor, auch wenn dies nicht immer offensichtlich ist. Es ist bereits der Fall, wenn in (sequentiellen) Programmen `asynchrone Ausnahmen` zu behandeln sind. Das Auftreten einer Ausnahme führt zu einer nicht beschriebenen Verzweigung im Steuerfluß des Programms. `Ereignisgesteuerte Anwendungen` sind ein weiteres Beispiel. Der Steuerfluß wird nicht im Programm festgelegt, sondern durch Interaktion des Benutzers bestimmt.
Auch in `verteilten Anwendungen` ist man mit dieser Problematik konfrontiert. Die Aktivitäten eines Programmes werden auf mehrere Rechner (Prozessoren) verteilt. Die Sequentialisierung dieser Aktivitäten würde zwar die Programmierung vereinfachen, ist jedoch oft nicht gewünscht,

um Leistungszuwachs durch Nutzung paralleler Abarbeitung zu erreichen. Daß der Programmierer eines `Multiprozessorsystems` mit mehreren Prozessen konfrontiert ist, ist offensichtlich.

Heutige (sequentielle) Programmiersprachen kennen zur höheren Strukturierung des Steuerflusses eines Programms das Konzept der *Prozedur*. Die genannten Klassen von Anwendungsprogrammen lassen sich jedoch nicht allein über das Prozedurkonzept beschreiben. Sie erfordern eine weitere Strukturierung in *Prozesse*. Die beteiligten Prozesse sind jedoch oft im Programm nicht sichtbar. Vielmehr wird der Ablauf eines Mehrprozeßsystems oft in Prozeduren mit "Seiteneffekten" implizit unterstellt (`fork()` aus der C-Bibliothek auf UNIX-Systemen, asynchrone RPC etc.). Die Komplexität mehrerer Prozesse läßt sich jedoch nicht dadurch reduzieren, daß sie mit einer sequentiellen Syntax (Prozeduraufruf) beschrieben wird. Sie ist real vorhanden und sollte deshalb in der Struktur von Programmen sichtbar (transparent=durchschaubar) sein. Eine explizite Strukturierung eines Systems in Prozesse erscheint deshalb wünschenswert. Im Programm ist dann sichtbar, welche Aktivität zu welchem Prozeß gehört.

In diesem Beitrag werden grundlegende Mechanismen vorgestellt, wie ein Prozeßsystem in der (imperativen) Programmiersprache C realisiert werden kann und wie die Prozeßstruktur in einem Programm sichtbar gemacht werden kann. Es muß dabei eine Abbildung auf die Elemente der Sprache erfolgen. Man kommt allerdings nicht umhin, eine kleine Menge von Prozeduren mit "Seiteneffekten" zu verwenden.
Die detaillierte Beschreibung der Umsetzung des in C benutzten Prozedur-Konzeptes auf Maschinenbefehls-Ebene ist die Basis für das Verständnis der Manipulation dieser Umsetzung, mit denen C-Funktionen als Coroutinen abgearbeitet werden können. Die vom Compiler erzeugten C-Funktionen werden durch Manipulation der Ablaufumgebung als Coroutinen abgearbeitet. Die Umschaltung zwischen Coroutinen ist der zentrale Mechanismus des vorgestellten Systems. Es werden dabei bewußt verschiedene Hardware- und Systemplattformen betrachtet (CISC-, RISC-Prozessoren, Betriebssysteme: UNIX, MSDOS), um die Allgemeingültigkeit der Konzepte nachzuweisen und die Auswirkungen verschiedener Architekturen zu untersuchen. Coroutinen sind durch zwei Kriterien aneinander gekoppelt. Zum einen ist im Programm der Umschaltpunkt (*TRANSFER*) festgelegt und zum anderen ist die Folge-Coroutine angegeben. Die Entkopplung von Coroutinen bzgl. dieser Kriterien führt in der Konsequenz zu unabhängigen Prozessen. Durch Verlagerung der Auswahl der Folge-Coroutine in eine zentrale Instanz (*Scheduler*) lassen sich Coroutinen entkoppeln und verschiedene Auswahlstrategien realisieren.
Unabhängige Prozesse enthalten keine Umschaltoperationen, wie Coroutinen. Löst man auch die Umschaltoperation aus den Programmen heraus, muß sichergestellt werden, daß die Umschaltoperation durch einen "äußeren Prozeß" veranlaßt wird und zum Entzug (*preemption*) der Steuerung im gerade abgearbeiteten Programm (Prozeß) führt.
Die ursprünglichen Coroutinen sind durch Herauslösen der Umschaltung und der Auswahl in einem Grade voneinander entkoppelt, daß man von unabhängigen Prozessen sprechen kann. Man bezeichnet diese Prozesse als *Leichtgewichtsprozesse* oder *Threads*, da sie sich alle auf einen gemeinsamen Adreßraum beziehen. Die vorgestellte Realisierung ist in fünf Schichten strukturiert, und wird schrittweise aufeinander aufgebaut:

1. Wechsel der Registerwerte der CPU
2. Coroutinen-Umschaltung (Kontext-Wechsel)
3. Auswahl der Folge-Coroutine (Scheduling)
4. Prozesse mit Entzug – Preemptive Threads
5. in Prozesse (Threads) strukturierte Anwendung

Es existieren bereits eine Vielzahl von Thread-Realisierungen auf verschiedenen Systemen. Diese stellen allesamt nur die Dienstschnittstelle bereit und lassen die interne Realisierung im Verborgenen. Durch Aufdecken interner Wirkmechanismen und deren (einfache) Realisierbarkeit soll das Verständnis von Mehrprozeßsystemen gefördert werden.