Die PURE-OSEK API
Die Spezialisierung einer objektorientierten Betriebssystem-Familie

Holger Papajewski, Wolfgang Schröder-Preikschat,
Olaf Spinczyk, Ute Spinczyk

30. November 2000

Zusammenfassung

In diesem Dokument präsentieren wir unseren Ansatz, wie ein OSEK-konformes Betriebssystem als spezielles Mitglied der Betriebssystemfamilie PURE implementiert werden kann. Mit PURE wenden wir das Konzept der Programmfamilien in Kombination mit objektorientierten Implementierungstechniken an.

Durch die Entwicklung eines OSEK-konformen Familienmitgliedes zeigen wir beispielhaft die Flexibilität und Wiederverwendbarkeit PURE's. Das resultierende System gleichzeitig eine Alternative zu einer „normalen" OSEK-Implementierung dar. Es wird variable Funktionalität sowie eine objektorientierte Schnittstelle geboten, indem die Konfigurationsmöglichkeiten des PURE-Systems genutzt werden.

1 Motivation

Um die Austauschbarkeit und Nutzbarkeit fremder Applikationen bzw. Systeme zu verbessern, wurde von den führenden Herstellern der Automobilindustrie der OSEK-Standard entwickelt. Die OSEK-Betriebssystem-Spezifikation [1] beschreibt die Schnittstellen eines Betriebssystems, welches auf die Bedürfnisse automotiver Systeme zugeschnitten ist. Bei Einhaltung der durch die Spezifikation vorgeschriebenen Schnittstellen ist ein unkomplizierter Austausch von Software zwischen unterschiedlichen Herstellern möglich.

Allerdings bringen fest definierte Schnittstellen auch Probleme mit sich. Oftmals fehlen für eine bestimmte Anwendung wichtige Fähigkeiten oder ungenutzte Eigenschaften liegen brach. Der daraus resultierende höhere Verbrauch von Betriebsmittel kompensiert die obigen Vorteile. In Anbetracht der enormen Anzahl der jährlich hergestellten Kraftfahrzeuge (weltweit 54.4 Millionen im Jahr 1997 [3]) und des Einsatzes mehrerer vernetzter Mikroprozessoren in jedem, muß eine Lösung des obigen Konflikts gefunden werden.

Hierzu definiert die OSEK-Spezifikation vier sogenannte Konformitätsklassen, welche mit vier unterschiedlichen Betriebssystemvarianten gleichzusetzen sind. Durch Auswahl einer dieser Varianten entscheidet die Anwendung, welche OSEK-Eigenschaften zur Laufzeit zur Verfügung stehen. Mit Blick auf die Konformitätsklassen definiert die OSEK-Spezifikation eine aus vier Mitgliedern bestehende Betriebssystemfamilie. Unserer Meinung nach ist dies ein Schritt in die richtige Richtung. Aber er ist nicht groß genug.

Bei der OSEK-Spezifikation besteht wie bei jedem anderen Standard auch ein generelles Problem, das aus den unterschiedlichen Anwendungsszenarien resultiert. Für die einen reichen die gebotenen Eigenschaften nicht aus. Für andere Anwender wird zu viel Funktionalität geboten. Gerade im Gebiet der automotiven Systeme, welche äußerst komplexe Spezialsysteme sind, kommt diesem Problem eine besondere Bedeutung zu. Der Systementwickler ist mit dem Wissen über die Anwendung und der verwendeten Hardware in der Lage, ein hoch effizientes, optimiertes System zu entwerfen und zu implementieren. Allerdings entspricht die Auswahl einer bestimmten OSEK-Konformitätsklasse eher einem allgemeinem System, welches für eine größere Anzahl von Anwendungen geeignet ist.

Unser Ansatz zur Lösung dieses Problems ist die Verwendung der Betriebssystemfamilie PURE mit feingranularen Konfigurationsmöglichkeiten. Der grundlegende Gedanke besteht darin, möglichst viel Funktionalität für unterschiedliche Anwendungsgebiete zur Verfügung zu stellen[1]. Dem Entwickler wird dann die Möglichkeit geboten, explizit die Eigenschaften auszuwählen, welche von seiner konkreten Applikation benötigt werden.

Natürlich bedeutet das nicht, daß für jede Applikation ein eigenes Betriebssystem entwickelt werden muß. Mit einem familienbasierten Design und einer objektorientierten Implementierung ist es möglich, das System so zu organisieren, daß kein doppelter Code benötigt wird.

Im folgenden möchten wir das Familienkonzept in Verbindung mit objektorientierter Implementierung, so wie wir es in PURE verwenden, vorstellen. Nach diesen fundamentalen Erläuterungen wird gezeigt, wie die Integration der OSEK-Schnittstellen in das bestehende System erfolgte. Letztendlich werden einige Ergebnisse und Pläne für die Zukunft PURE's präsentiert.

2 Die Betriebssystemfamilie PURE

Die Idee PURE's ist es, ein Betriebssystem als eine objektorientiert[9] implementierte Programmfamilie[8] zu sehen. Programmfamilien helfen beim Entwurf eine monolithische Programmstruktur zu verhindern. Die Objektorientierung ermöglicht eine effiziente Implementierung der hoch modularen Systemstruktur.

2.1 Inkrementeller Systementwurf

Das Konzept der Programmfamilien schreibt keine bestimmte Art der Implementierung vor. Eine sogenannte „minimale Menge von Systemfunktionen" ergibt eine Plattform mit fundamentalen Abstraktionen, welche durch „minimale Systemerweiterungen" ausgebaut wird. Diese Erweiterungen basieren auf einem inkrementellen Systementwurf [7]. Dabei stellt jede neue Ebene die minimale Basis für zusätzliche funktionale Erweiterungen des Systems dar. Wesentliche Entwurfsentscheidungen werden so weit wie möglich hinausgezögert, um das System nicht unnötig einzuschränken. Die Systemkonstruktion findet „bottom-up" statt, wird aber „top-down" (anwendungsgetrieben) kontrolliert. Die eigentliche Anwendung stellt somit letztendlich die finale Systemerweiterung dar. Die traditionelle Grenze zwischen System und Applikation entfällt. Das Betriebssystem geht in die Anwendung über und umgekehrt.

Die Vererbung ist eine geeignete Technik, um neue Systemerweiterungen einzuführen bzw. existierende durch alternative Implementierungen zu ersetzen. Wie auch

[1] Das ist bisher nicht der Fall, aber die Funktionalität nimmt stetig zu.

immer, die Systemerweiterungen werden hinsichtlich der speziellen Bedürfnisse des Benutzers angepaßt und stehen in Koexistenz mit der Anwendung zur Laufzeit zur Verfügung. Die Applikation muß nicht für (Betriebssystem-) Ressourcen bezahlen, welche sie nicht nutzt.

2.2 Wiederbelebung der Programmfamilien

Die konsequente Anwendung des Familienkonzeptes während des Software-Entwurfes führt zu einer hoch modularen Struktur. Neue Systemeigenschaften werden zu dem bereits bestehenden System hinzugefügt. Da eine starke Analogie zwischen den Begriffen „Programmfamilie" und „Objektorientierung" besteht (Abbildung 1), ist es naheliegend, Programmfamilien objektorientiert zu konstruieren [4]. Die Parallelen beider Ansätze sind deutlich sichtbar. Die minimale Basis von Systemfunktionen des Familienkonzeptes entspricht den Basisklassen der Objektorientierung. Minimale Systemerweiterungen werden durch Ableitungen der Basisklassen eingeführt. Die Vererbung und der Polymorphismus ermöglichen die Koexistenz unterschiedlicher Implementierungen gleicher Schnittstellen. Die Wiederverwendbarkeit von Programmcode wird signifikant erhöht, da verschiedene Familienmitglieder sich gemeinsame Eigenschaften teilen.

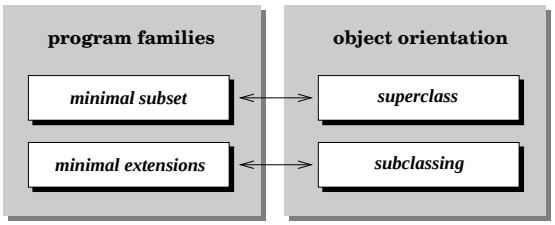

Abbildung 1: Programmfamilien und Objektorientierung

Im Bereich der eingebetteten/parallelen Systeme muß Vererbung auch über Adressräume, Knoten und Netzwerkgrenzen anwendbar sein [5]. Dies ist ein bedeutender Gesichtspunkt für die Entwicklung PURE's. Das Familienkonzept ist das grundlegende Entwurfsprinzip. Die Objektorientierung wird hauptsächlich zur Implementierung und nicht zum Entwurf genutzt.

2.3 Betriebssystem-Baukasten

PURE ist ein „offenes" Betriebssystem. Alle Abstraktionen sind dem Systementwickler oder Anwendungsprogrammierer sichtbar. Das Gesamtsystem besteht aus einer Bibliothek aus kleinen, „handlichen" Objektmodulen. Diese Module sind hinsichtlich der Anzahl der in ihnen enthaltenden Referenzen zu Funktionen bzw. Variablen klein. Damit werden Binder unterstützt, schlanke Betriebssysteme zu erzeugen, die nur Komponenten beinhalten, welche durch die Anwendung genutzt (bzw. referenziert) werden. Als Voraussetzung wird eine hoch modulare Systemarchitektur benötigt, welche durch den familienbasierten Entwurf und die objektorientierte Implementierung geschaffen werden.

PURE hat vieles gemeinsam mit OSKit [6]. Aber anstatt eine neue System-Architektur zu erfinden, ermöglicht PURE die Konstruktion von unterschiedlichen Architekturen. Durch PURE wird keine bestimmte Betriebssystem-Architektur vorge-

schrieben. Vielmehr wird ein Baukasten zur Entwicklung von Betriebssystemen zur Verfügung gestellt. Ob nun ein monolithisches oder ein mikrokern-basiertes System erstellt werden soll, liegt nur am Benutzer, der die Elemente PURE's zur Systementwicklung entsprechend seiner Vorgaben nutzt. Die Vorgaben für ein maßgeschneidertes Betriebssystem kommen von der Anwendung selbst.

2.4 Die Nucleus Familie

Entsprechend den Anforderungen der Anwendung sehen die den Nucleus bildenden Einheiten unterschiedlich aus. Jede dieser Konfigurationen repräsentieren ein Mitglied der Nucleus-Familie. Ein Auszug des Nucleus-Familienbaums ist in Abbildung 2 zu sehen. Das Bild zeigt sechs Familienmitglieder, wobei jedes einen speziellen Betriebs-

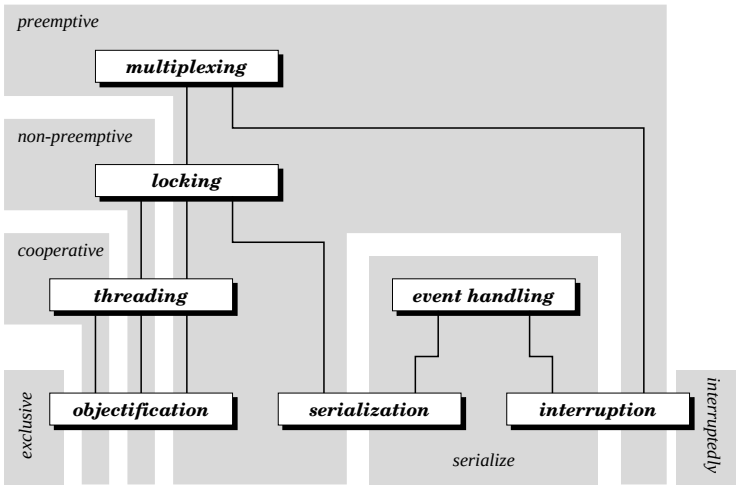

Abbildung 2: Der Nucleus-Familienbaum

modus implementiert. Jedes Mitglied besteht aus einem oder mehreren Funktionsblöcken und ist durch deren funktionalen Hierarchie beschrieben. Die Funktionsblöcke können dabei in unterschiedlichen Konfigurationen wiederverwendet werden. Um die Familie nicht einzuschränken, wurden Entwurfsentscheidungen weit hinausgeschoben und durch höhere Abstraktionen gekapselt. Damit kann PURE flexibel angepaßt werden, wie die folgenden Szenarien zeigen:

1. Eine grundlegende Betriebsart PURE's verarbeitet nur Unterbrechungen *(interruptedly)*. Dieses Familienmitglied unterstützt lediglich low-level Trap/Interrupt-Bearbeitung. Es existieren keine Thread-Abstraktionen. Einzig die Ausführung von Unterbrechungsbehandlungen wird unterstützt (*interruption*).

2. Um asynchron initiierte Aktionen der Unterbrechungsbehandlung mit der synchronen Abarbeitung des unterbrechenden Programmes abzustimmen (*reconcile*), wurde Szenario 1 minimal erweitert. Das entstandene Mitglied sichert die interrupt-transparente (*serialization*), synchrone Bearbeitung von Ereignissen (*driving*).

3. Die zweite Grundbetriebsart ist nur in der Lage, ein einzelnes aktives Objekt auszuführen (*exclusive*). In dieser Konfiguration kann der Nucleus nur einen einzigen Thread aufsetzen (*objectification*). Das gesamte System wird durch die Anwendung kontrolliert, wobei die Anwendung als Spezialisierung eines aktiven Objektes angesehen wird. Sie ist das einzige aktive Objekt im System.

4. Eine weitere minimale Erweiterung zu 3 führt zum kooperativen Thread-Scheduling (*cooperaive*). Es werden hier keine Entwurfsentscheidungen gefällt, außer daß die Threads durch aktive Objekte repräsentiert werden und der Wechsel zwischen diesen durch die Anwendung gesteuert wird (*threading*). Es können beliebig viele aktive Objekte im System vorhanden sein.

5. Die Erweiterung der Funktionen für Thread-Scheduling um serialisierte Ausführung (*locking*), ermöglicht die nicht-präemptive Abarbeitung (*non-preemptive*) von aktiven Objekten in einer interrupt-gesteuerten Umgebung. Das Thread-Scheduling ist weiterhin kooperativ. Der Nucleus ist allerdings in der Lage, Threads aus der Unterbrechungsbehandlung heraus zu schedulen. Durch Unterbrechungsbehandlungen ausgeführte Aktionen mit globalen Auswirkungen sind entsprechend zu synchronisieren (*serialization*).

6. Das Unterbrechungs-gesteuerte Multiplexen der CPU zwischen Threads ermöglicht die autonome, präemptive Abarbeitung (*preemptive*) von aktiven Objekten. In diesem Fall wird der Nucleus um Treiber-Module erweitert (*multiplexing*), welche das zeitgesteuerte Thread-Scheduling übernehmen.

2.5 Analyse

Das PURE-System ist in C++ implementiert und auf i80x86-, sparc-, alpha-, m68k-, ppc60x-, C167-, AVR- und ARM-basierenden Plattformen portiert. Derzeit besteht der Nucleus aus über 100 Klassen, welche mehr als 600 Methoden exportieren. Jede Klasse implementiert dabei einen abstrakten Datentyp. So besteht z.B. der Thread-Kontroll-Block aus 45 Klassen, die eine 14 Ebenen umfassenden Hierarchie aufbauen. Damit stellt sich die Frage nach den Kosten in Bezug auf Speicherverbrauch und Geschwindigkeit.

2.5.1 Speicherverbrauch

Wie die Tabelle 1 zeigt, lassen sich trotz der hoch-modularen Struktur des Nucleus kleine und kompakte Systeme erzeugen. Als Basis für die Messung dienten die oben

family member	**size** (in byte)			
	text	*data*	*bss*	*total*
interrupedly	812	64	392	1268
serialize	1882	8	416	2306
exclusive	434	0	0	434
cooperative	1620	0	28	1648
non-preemptive	1671	0	28	1699
preemptive	3642	8	428	4062

Tabelle 1: Speicherverbrauch PURE's

beschriebenen Mitglieder der Nucleus-Familie (Abbildung 2). Die Ergebnisse wurden mittels des GNU g++ 2.7.2.3 für i586 unter Red Hat Linux 5.0 ermittelt.

2.5.2 Leistungsverhalten

Die Abbildung 3 zeigt die Anzahl der CPU-Takt-Zyklen (i586) während der Abarbeitung einer Ausnahmebehandlung. Die Taktfrequenz während des Experimentes betrug 166 MHz. Die Messung erfolgte mittels des Zähler-Registers des Pentium-Prozessors. Es wurde wiederum der oben beschriebene C++ Compiler verwendet. Zum Zeitpunkt

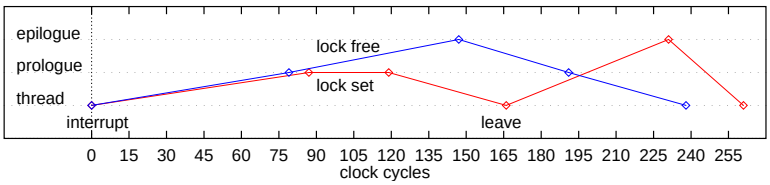

Abbildung 3: Latenzzeiten der Unterbrechungsbehandlung

0 wird ein laufender Thread unterbrochen. Die Abarbeitung des asynchronen Teils der Unterbrechungsbehandlung (der sogenannte „Prologue") startet nach 79 bzw. 87 Taktzyklen. In Abhängigkeit der Anwendung und der Konfiguration des Nucleus kann die gesamte Bearbeitung der Unterbrechung im Prologue oder aber in einem zweiten Teil, genannt Epilogue, erfolgen. Der Epilogue wird synchronisiert zu den Aktivitäten des Nucleus ausgeführt. Hierzu stehen unterschiedliche Locking-Mechanismen zur Verfügung. Der unterschiedliche Zeitpunkt der Prologue-Aktivierung ist durch die Verwendung verschiedener Programm-Instrumentierungen für die *lock-free* und *lock-set* Szenarien zu erklären.

Die Zahlen zeigen, daß abhängig vom Zustand der Sperre, die normale Programmausführung für 238 bzw. 261 Taktzyklen unterbrochen wird. Die Ausführung des Epilogues beginnt beim 147. bzw. 231. Takt. Der Unterschied von 84 Zyklen kennzeichnet die kürzeste Verzögerung der Ausführung des Epilogue, wenn die Sperre gesetzt ist. Weitere Verzögerungen hängen von zwei Faktoren ab: (1) die Zeit, die benötigt wird, um den geschützten Bereich zu verlassen, und (2) die Anzahl der vorangehenden Epilogue in der Liste. Das Hinauszögern eines Epilogues (Takt 87-119) benötigt 32 Takte. Die Aktivierung eines verzögerten Epilogues (Takt 166-231) erfolgt in 65 Takten. In beiden Aktionen wird die Unterbrechungs-transparente Synchronisation der Warteliste durchlaufen. Nach der Ausführung des Prologues erfolgt die Rückkehr zur unterbrochenen Anwendung (Takt 119-166 und 191-238) in 47 Taktzyklen. Dabei ist es egal, ob der Thread innerhalb oder außerhalb eines geschützten Bereiches unterbrochen wurde. Die Rückkehr vom Epilogue (Takt 147-191 und 231-261) kostet 44 bzw. 30 Takte. Die Differenz ergibt sich aus den unterschiedlichen Anweisungen, welche für den gesperrten bzw. den nicht gesperrten Fall nötig sind.

Die Zeiten für das Schedulen der Threads hängen vom verwendeten Nucleus Familienmitglied (Abbildung 2) ab. Sie liegen in einem Bereich von 49 Takten (*cooperative*) bis 300 Taktzyklen (*preemptive*). Dies alles zeigt die „leichtgewichtige" Struktur PURE's, obwohl eine große Anzahl von Abstraktionen (Klassen, Module, Funktionen) beteiligt sind. Aber: „*It is the system design which is hierarchical, not its implementation*" [7].

3 OSEK Integration

3.1 Konzept

Die OSEK API ist durch C-Funktionen beschrieben. Pure ist allerdings ein objektorientiertes System mit Klassen, Objekten und Methoden. Um eine spätere Erweiterung der OSEK-Implementierung zu erlauben und die Wiederverwendbarkeit des Quellcodes in beiden Richtungen zu gestatten, entschlossen wir uns, den Entwurfskonzepten Pure's entsprechend, die OSEK-Erweiterung objektorientiert zu gestalten. Die Abbildung von den C zu den C++ Schnittstellen erfolgt mittels Makros sowie eines kleinen Bibliothek (Abbildung 4).

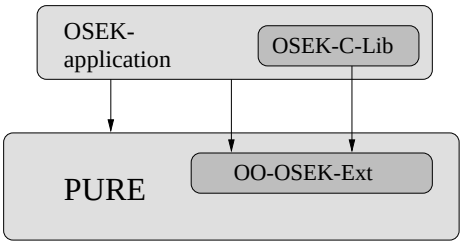

Abbildung 4: OSEK Anwendungs-Schnittstellen

Diese Vorgehensweise ermöglicht einer objektorientierten OSEK-Anwendung den direkten Zugriff auf die OO-OSEK API. Gleichzeitig lassen sich weitere Eigenschaften PURE's nutzen.

OSEK mit seinen Konformitätsklassen und verschiedenen Scheduling-Strategien wird als eine Unterfamilie von PURE angesehen. Die Konfiguration von OSEK und Pure vollzieht sich mittels der gleichen Mechanismen. Zur Unterstützung der Konfiguration für OSEK-Anwendungen wurde eine Werkzeug geschaffen, welches die „Familienmitglieder-Schalter" entsprechend der gewählten OSEK-Konfiguration setzt.

Der folgende Abschnitt demonstriert die Integration OSEK's beispielhaft an der Thread-Hierarchie.

3.2 Fallstudie: OSEK-Threads

Die Abbildung 5 zeigt einen Teil der PURE Klassenhierarchie. Die Pfeile kennzeichnen eine Ableitungsbeziehung zwischen den Klassen. Die Rechtecke mit unterbrochener Linie stellen Konfigurationspunkte dar, an welchen unterschiedliche Systemeigenschaften ausgewählt werden können.. So kann der `Trimmer` z.B. als normale Klasse verwendet werden, wobei er allerdings durch den `Customer` oder den `Conductor` implementiert ist. Welche der beiden Alternativen zum Einsatz kommt, ist abhängig von der jeweiligen Konfiguration des Systems. Die Thread-Hierarchie kapselt den Zustand von PURE-Threads (Prozeß-Kontroll-Block) und die dazugehörigen Zugriffsfunktionen.

Am Anfang der Hierarchie steht die `Activity`-Klasse, welche eine sehr einfache Abstraktion einer Koroutine darstellt. Diese und alle ihre Basisklassen werden von OSEK wiederverwendet. Die `Visitor`-Klasse ist von `Activity` abgeleitet. Sie nutzt die Mehrfachvererbung um den Prozeß-Kontroll-Block um ein Verkettungsglied zu erweitern. Damit können die Threads in eine Ready-Liste aufgenommen werden.

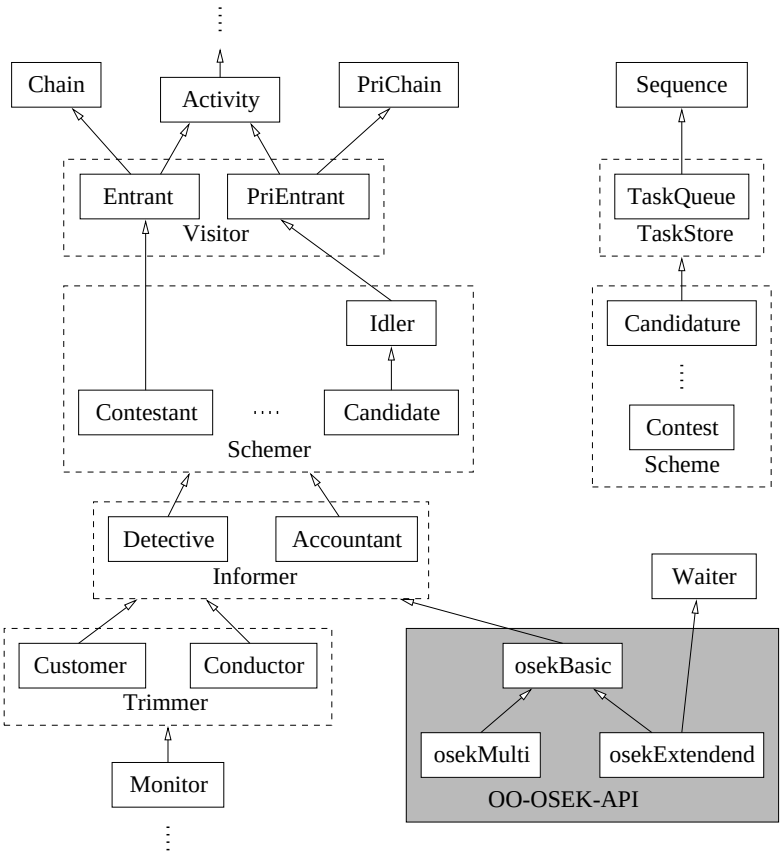

Abbildung 5: Thread-Hierarchie

Das OSEK-Thread-Scheduling basiert auf Prioritäten. Folglich ist die Ready-Liste eine priorisierte Liste und dem Verkettungselement ist eine Priorität zugeordnet. PURE besaß bisher diese Funktionalität nicht. Allerdings ist prioritätsbasiertes Scheduling eine wichtige und oft genutzte Fähigkeit. So entschieden wir, eine Abstraktion für priorisierte Listen zu implementieren. Außerdem wurde die `Visitor`-Abstraktion zu einem Konfigurationspunkt. Wird z.B. nur einfaches Round-Robin benötigt, so ist der `Visitor` von `Chain` abgeleitet. Für prioritätsbasiertes Scheduling wird `PriChain` als Basis verwendet, welches ein Element einer priorisierten Liste darstellt.

Die `Schemer`-Abstraktion erweitert den `Visitor` um schedulingspezifische Informationen und Methoden. Sie ist ebenfalls ein Konfigurationspunkt, an welchen eine von den fünf derzeit existierenden Scheduling-Strategien gewählt werden kann. Die Auswahl des `Schemer` muß mit der der `Scheme`-Abstraktion, welche den Scheduler darstellt, übereinstimmen. Für OSEK implementierten wir eine neue, prioritätsbasierte Scheduling-Strategie, welche auch komplett von nicht-OSEK Anwendungen genutzt werden kann.

Die konfigurierbare `Informer`-Klasse bietet Methoden zur Abfrage des Thread-Zustands an. Nach dieser Klasse verzweigt sich die Hierarchie in zwei Richtungen. Der eine Teil bildet die Standard-PURE-Threads aus. Der andere beinhaltet OSEK-spezifische Thread-Erweiterungen. Dieser Teil implementiert die objektorientierte

OSEK API.

3.3 Ergebnisse

Unsere familienbasierte OSEK-Implementierung bietet die gesamte OSEK API an. Alle Konformitätsklassen und Scheduling-Strategien werden unterstützt. Einzigst der erweiterte Fehler-Modus wird nicht unterstützt. Die durch das OSEK-Komitee bereitgestellten Konformitätstest [2] bestätigen die volle Kompatibilität unserer PURE/OSEK-Implementierung.

OSEK-Systeme haben eine statische Struktur, welche durch eine OIL[2]-Konfigurationsdatei beschrieben wird. Zusätzlich zur OSEK-Laufzeit-Unterstützung entwickelten wir ein Werkzeug namens `OilGen`, welches OIL-Beschreibungen versteht und PURE entsprechend konfiguriert. Für einen Anwendungsprogrammierer besteht kein Unterschied zwischen unserer familienbasierten PURE/OSEK-Implementierung und anderen OSEK-Systemen. Allerdings bieten wir zusätzlich eine objektorientierte Schnittstelle sowie zusätzliche PURE-Funktionalitäten, wie z.B. weitere Synchronisationsmechanismen.

Tabelle 2 zeigt den Vergleich von zwölf verschiedenen PURE-Systemen, inklusive der drei bedeutenden OSEK-Konfigurationen. Sie beinhaltet die Größe der Code- und Daten-Bereiche sowie die Kontextwechselzeit für eine äquivalente Anwendung, in welcher sich zwei Threads gegenseitig synchronisieren.

scheduling	synchronization	text	data	context switch
FCFS-thread	eventbox	2871	1052	94
	signalbox	3351	1076	141
	cooperative	2095	1052	61
FCFS-bundle (different)	eventbox	3371	1052	154
	signalbox	3863	1076	159
	cooperative	2643	1052	78
FCFS-bundle (same)	eventbox	3391	1052	126
	signalbox	3879	1076	142
	cooperative	2667	1052	62
cooperative	events	3242	2248	148
preemptive	events	3674	2248	202
mixed	events	3922	2248	218

Tabelle 2: PURE vs. OSEK

Die Programmgrößen wurden mittels der 80x86 Implementierung (Compiler `egcs-1.0.2`) PURE's ermittelt. Die Kontextwechselzeiten sind als Taktzyklen eines Pentium II Prozessors angegeben.

Der obere Teil der Tabelle enthält die Größen der PURE-Version des Testprogramms mit unterschiedlichen Scheduling-Strategien und Synchronisations-Mechanismen. *FCFS-thread* ist einfaches Round-Robin-Scheduling. Ein `Bundle` ist eine Gruppe von Threads, wobei sowohl zwischen den Threads innerhalb einer Gruppe, als auch zwischen den Gruppen gewechselt werden kann. Unabhängig vom gewählten Synchronisations-Mechanismus benötigt die einfache Scheduling-Strategie etwa 500 Byte weniger als eine der zwei Gruppen-Versionen. Die Zeit für einen Kontextwechsel

[2] open implementation language

ist nur höher, wenn zwischen Threads unterschiedlicher Gruppen gewechselt werden muß.

Sowohl die Größe des Programmcodes als auch die Zeit für den Kontextwechsel hängen stark vom verwendeten Synchronisations-Mechanismus ab. Die „cooperative"-Variante ist die billigste. Hierbei findet ein unbedingter Wechsel vom aktiven Thread zum nächsten bereiten Thread statt. Eine `Signalbox` ist ein Synchronisations-Objekt, welches eine Liste mit wartenden Threads verwaltet. Ihr Verhalten läßt sich mit einer zählenden Semaphore vergleichen. Die Synchronisation mittels der `Eventbox` ist im Vergleich zur `Signalbox` billiger. Sie entspricht dem OSEK-Event-Mechanismus. Die `Eventbox` bietet weniger Flexibilität als die `Signalbox`, aber sie benötigt dafür auch keine Liste für wartende Threads. Für den `Eventbox`-Mechanismus werden nur zwei Bitfelder innerhalb des Prozeß-Kontroll-Blockes verlangt.

Der untere Teil der Tabelle zeigt die OSEK-Versionen der Test-Anwendung. Zu der Größe des Daten-Bereichs muß gesagt werden, daß dieser die Stacks für die zwei Anwendungs-Threads enthält, wobei jeder 1024 Byte groß ist. Somit verringert sich der reale Speicherverbrauch für die Prozeß-Kontroll-Blöcke und die globalen Variablen auf 200 Byte. In PURE kann der initiale Thread den Boot-Stack benutzen, so daß nur ein Stack von 1024 Byte im Daten-Bereich benötigt wird. Die verbleibenden 28 bzw. 52 Bytes beinhalten aber keine Prozeß-Kontroll-Blöcke. Diese sind dynamisch auf dem Stack des initialen Threads reserviert worden. Sie benötigen zusammen 112 Byte. Somit stehen 200 Byte der OSEK-Variante 140 bzw. 164 Byte der reinen PURE-Variante gegenüber.

Die OSEK-Scheduling-Strategien benötigen mehr Code und Zeit, als die des reinen PURE-Systems. Priorisierte Listen, die Möglichkeit den Scheduler zu sperren sowie die Mehrfachaktivierung sind nicht kostenlos zu erhalten.

4 Schlußfolgerungen

Mit diesem Beitrag haben wir unseren Ansatz, objektorientierte Betriebssysteme effizient für den eingebetteten Bereich zu implementieren, aufgezeigt. Die Integration der standardisierten OSEK-Schnittstellen in unser Forschungssystem PURE zeigt die Flexibilität der objektorientierten Implementierungs-Techniken in Kombination mit einem familienbasiertem Entwurf. Es war uns möglich, OSEK-Eigenschaften so in PURE zu integrieren, daß sie auch durch andere nicht-OSEK Anwendungen nutzbar sind. Auf der anderen Seite profitiert die OSEK-Implementierung stark von den vorhandenen PURE-Abstraktionen.

Ein zweiter wesentlicher Erfolg besteht darin, daß die Größe und das Laufzeitverhalten des objektorientierten Betriebssystems dessen Einsatz für Anwendungen im Bereich tief eingebetteter Systeme erlauben. Bei einer Größe von einem halben kByte bis zu ein paar kByte bleibt genügend Spielraum für die Implementierung der Anwendungsalgorithmen übrig.

Literatur

[1] OSEK/VDX Steering Committe. OSEK/VDX Operating System, October 1997. Version 2.0 Revision 1

[2] OSEK/VDX Steering Committe. OSEK/VDX OS Test Procedure, April 1999, Version 2.0

[3] contact. Journal of the German-French Chamber of Industry and Commerce, September 1998. ISSN 1161-7403.

[4] J. Cordsen, W. Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (I-WOOOS '91)*, 24-28, Palo Alto, CA, 1991.

[5] J. Nolte, W. Schröder-Preikschat. Dual Objects - An Object Model for Distributed System Programming. In *Proceedings of the Eighth ACM SIGOPS European Workshop, Support for Composing Distributed Applications*, 1998. http://www.acm.org/sigops/EW98/papers.html.

[6] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In Proceedings of the Sixteenth ACM Symposium on Operating System Principles, 38-51, Saint-Malo, France, 1997.

[7] A. N. Habermann, F. Flon, L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266-272, 1976.

[8] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *Transaction on Software Engineering* SE-5(2), 1979.

[9] P. Wegner. Classifications in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173-182, 1986.