Präemptives Planungswesen für aktive und passive Objekte

Studienarbeit

Michael Schulze

„Otto-von-Guericke“ Universität Magdeburg
Fakultät für Informatik
Institut für Verteilte Systeme

Aufgabenstellung: Prof. Dr.-Ing. Wolfgang Schröder-Preikschat

Betreuung: Dipl. Inf. Friedrich Schön

GMD - Forschungszentrum
Informationstechnik GmbH
Institut für Rechnerarchitektur und Softwaretechnik
Rudower Chaussee 5
12489 Berlin

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Im Rahmen dieser Arbeit erfolgte die Erweiterung der PURE-Betriebssystemfamilie, um ein präemptives Planungswesen für aktive und passive Objekte.

Die objektorientierte Betriebssystemfamilie PURE stellt verschiedene Familienmitglieder zur Verfügung, die in tiefsten, eingebetteten Systemen ihre Verwendung finden. PURE steht hierbei für Portable Universal Runtime Executive. Diese Programmfamilie [7] kann dabei auf unterschiedlichste Weise konfiguriert werden, so daß jede Anwendung nur den Funktionsumfang bekommt, den sie auch tatsächlich benötigt.

Das präemptive Planungswesen legt fest, zu welcher Zeit ein Objekt gestartet wird, unabhängig von der Ausprägung als passives oder aktives Objekt. Allein an der Priorität des Objektes wird über die Abarbeitung entschieden. Das präemptive Verhalten beschreibt einen sofortigen Wechsel zu einem anderen Objekt, sobald dieses höher priorisiert ist. Dieses Objekt hat ein Vorrecht auf die CPU.

Eine ereignisgesteuerte Verarbeitung wird angestrebt. Somit soll, sobald ein Ereignis stattgefunden hat und nachdem die hardwareseitigen Arbeiten abgeschlossen sind, der Scheduler aktiviert werden. Hierdurch kann auf die Objekte, welche nach Abarbeitung verlangen, umgehend reagiert werden. Das sofortige Reagieren auf ankommende Ereignisse soll in der Echtzeitfähigkeit des Systems münden. In wie weit das System harten oder weichen Echtzeitanforderungen, genügt wird zu untersuchen sein.

Weiterhin sind in dem zu modellierenden System Aufgaben der Synchronisation zu lösen. Passive Objekte könnten beispielsweise eine Spezialisierung einer Ausnahmesituation[1] und ein aktives Objekt z.B. einen Prozeß darstellen. Da hier Mechanismen aus unterschiedlichen Bereichen des Betriebssystemes aufeinander treffen, bedarf die gemeinsame Behandlung der Synchronisation. Des weiteren wird die Priorisierung der Objekte vorgenommen, welches auch zu Problemen führen kann. Durch eine geeignete Behandlung kann jedoch dem Problem der Prioritätsumkehr entgangen werden.

[1] Ausnahmesituation - Interrupts/Traps

1.1 PURE

PURE ist eine Weiterentwicklung der Betriebssystemfamilie PEACE [11]. PEACE wurde an der GMD-FIRST (Berlin-Adlershof), für die Parallelrechner SUPRENUM und MANNA, entwickelt. PURE setzt aber hier, im Gegensatz zu PEACE, mehr auf das dynamische Rekonfigurieren. Daher steht vielmehr die „Skalierbarkeit nach unten“ im Vordergrund. Als Ergebnis entsteht eine feingranulare Klassenhierarchie, die Konfigurationen selbst kleinster, maßgeschneiderter Betriebssysteme gestattet, die auch in Bereichen extremer Resourcenknappheit einsetzbar sind [9]. Prinzipiell sind jeder Anwendung nur die Betriebssystemfunktionen bereitzustellen, die diese auch tatsächlich benötigt. Ist diese Funktionalität noch nicht Bestandteil der Klassenbibliothek, so wird diese erstellt und der Programmfamilie hinzugefügt. Nunmehr legt somit die Anwendung fest, welche Bestandteile im endgültigen System enthalten sind.

Die momentan bereitstehende C++ Klassenbiliothek wird in der Lehre und Forschung an der „Otto-von-Guericke“ Universität Magdeburg sowie in der Forschung und Entwicklung an der GMD-FIRST eingesetzt [14]. Diese unterstützt das kontrollierte Ausführen von Programmfäden, Synchronisation und Kommunikation zwischen diesen sowie Behandlung und Synchronisation von Unterbrechungen. Die PURE-Klassenbibliothek kann auch oberhalb von Unix (SunOS, Solaris, Linux) als Gastebenenversion auf den Prozessoren 80x86, sparc und ppc weiterentwickelt werden. Eine native Implementierung gibt es zur Zeit für die Prozessorarchitektur 80x86, PPC und den Siemens C167 Microcontroller. Spezielle Hardwareeigenschaften werden in Klassen eingebettet, so daß bei einer Portierung nur wenige spezielle maschinenabhänige Klassen neu erstellt werden müssen. PURE ist bis auf einige Zeilen Assembler- und C-Code [8, 4] durchweg in C++ [6, 2] objektorientiert implementiert.

1.2 Familienkonzept und Objektorientierung in PURE

Der Begriff Familienkonzept impliziert schon, daß es sich bei PURE nicht nur um ein Betriebssystem handelt, sondern um eine Betriebssystemfunktionalitätensammlung, welche konfiguriert werden kann, um die unterschiedlichsten Ausprägungen erzeugen zu können. Vom Bereich der minimalen Betriebssystemkerne für den Einsatz in tiefst eingebetteten Systemen, bis hin zu Mehrprozessorsystemen mit verteiltem Speicher, sollen Systeme generierbar sein.

Ausgehend von einer minimalen Basis werden immer neu Funktionalitäten hinzugefügt. Erweiterungen sollten immer so klein wie möglich sein, denn sie stellen jeweils eine neue Eigenschaft zur Verfügung. Ein System aus vielen dünnen Schichten entsteht, wobei von der Schicht n-1 die Komplexität zur Schicht n zunimmt. Jede Schicht für sich ist auch weiterhin einsetzbar, weil nur Methoden der Bestandteile unter ihr benutzt werden. Somit handelt es sich bei jeder weiteren Ebene wieder um eine minimale Basis.

Entwurfsentscheidungen sollten aus diesem Grund soweit wie möglich auf einen späteren Zeitpunkt verschoben werden [12]. Anwendungen werden durch

jenes Konzept nicht mehr als nötig eingeschränkt.

Eine Implementierungsart ist bei einem Familienkonzept nicht vorgeschrieben. Eine objektorientierte Implementierung ist jedoch hervorragend dafür geeignet. Die minimale Basis steht hier für die Basisklassen und eine funktionale Erweiterung wird durch abgeleitete Klassen dargestellt. Diese erweiterten Basisklassen stellen einen eigenen abstrakten Datentyp dar, der wiederum als minimale Basis mit erweitertem Funktionsumfang gesehen werden kann. So können etwa verschiedene Implementierungen für eine Klasse existieren. Die Anwendung kann dann durch Konfiguration entscheiden, welche der unterschiedlichen Ausprägungen sie einsetzen möchte.

Entwurfsentscheidungen können durch die Verwendung des dynamischen Bindens weit hinaus gezögert werden, so daß schließlich die Anwendung die Chance hat, eine geeignete Implementierung in das System einzugliedern.

1.3 Aktive vs. passive Objekte

Ein aktives Objekt kennzeichnet einen eigenen Kontrollfluß, dieser kann direkt auf einem Prozessor abgearbeitet werden. Um in einem System mehrere Kontrollflüsse parallel oder pseudo parallel ausführen zu können, besteht die Notwendigkeit, jedem einen eigenen Kontext zu geben. Der Laufzeitkontext beschreibt den augenblicklichen Zustand des Prozesses. Bei einem Wechsel von einem Laufzeitkontext zu einem anderen wird der Austausch der Kontexte vollzogen. Dadurch läuft ein anderer Kontrollfluß auf der CPU.

Ein passives Objekt ist hingegen eine Prozedur, die auf einem unterbrochenen Prozeß abläuft. Dieser Prozeß oder auch das aktive Objekt stellt für die Dauer der Abarbeitung seinen Kontrollfluß zur Verfügung. Es besteht aber keine feste Kopplung zwischen einem passiven Objekt und einem speziellen Kontrollfluß.

Die Ähnlichkeit zu Unterbrechungen ist zu erkennen. Daher ist z.B. ein Einsatzgebiet der passiven Objekte als Spezialisierung von Ausnahmesituationen zu fungieren.

Kapitel 2

Motivation

2.1 Stand der Kunst

In Betiebssystemen arbeiten Systemfunktionen und Interrupthandler oftmals an den gleichen Datenstrukturen. Damit die Daten nicht inkonsistent werden, muß für eine Synchronisation gesorgt sein. Ein globales Verbot von Interrupts während der Ausführung von Systemfunktionen wäre denkbar (in vielen Betriebssystemen so realisiert [1]), ist aber ungünstig, wenn kurze Reaktionszeiten benötigt werden. Weiterhin könnten auch Ereignisse verloren gehen, auf die nicht reagiert werden kann. Bei kritischen Systemen wie etwa Steuerungsanlagen von Chemiebetrieben, hätte ein verlorenes Ereignis fatale Folgen.

Die Unterbrechungsbehandlung wird daher oftmals in zwei Phasen aufgeteilt. Ein Modell nach dem das BSD Unix Betriebssystem arbeitet, nennt sich Bottom Half und Top Half [5].

In der Betriebssystemfamilie PURE wird die Interruptbehandlung ebenfalls in zwei Phasen aufgeteilt:

1. Prolog: Interrupt-Vorbehandlung
2. Epilog: Interrupt-Nachbehandlung

Dieses Prinzip nennt sich Schleusen.

Der Prolog dient der Reaktion auf den Hardwareinterrupt. Während seiner Ausführung kann dieser nur durch Interrupts höherer Ebenen unterbrochen werden. Zu den Nukleusaktivitäten läuft der Prolog asynchron. Von einer Konsistenz der Datenstrukturen kann zu diesem Zeitpunkt somit nicht ausgegangen werden. Aus diesem Grunde ist es untersagt, in der Prologphase zentrale Datenstrukturen, wie etwa die Readyliste, zu manipulieren.

Die zweite Phase der Interruptbehandlung, der Epilog, läuft synchronisiert zu den Systemaufrufen ab. Daher können hier auch globale Daten verändert werden. Interrupts sind bei der Abarbeitung von Epilogen grundsätzlich zugelassen.

Um das beschriebene Verhalten erfolgreich zu realisieren, müssen die Unterbrechungsbehandlungsroutinen in zwei einzelne Funktionen aufgesplittet werden. Im Folgenden muß der Kern des Systems auf kritische Abschnitte untersucht werden. Sind solche Bereiche identifiziert, wird eine Schloßvariable

eingeführt, die Aufschluß gibt, ob gerade ein Systemaufruf oder ein Epilog in Ausführung ist. Das Setzen der Schloßvariablen erfolgt beim Betreten des kritischen Abschnittes. Beim Verlassen wird diese wieder gelöscht. Weiterhin muß eine Epilogliste eingeführt werden, die auszuführende Epiloge aufnimmt, wenn der kritsche Bereich belegt ist. Diese Liste arbeitet nach dem FIFO[1]- Prinzip. Prologe dürfen ihren Epilog starten, wenn die Schloßvariable nicht gesetzt ist, anderenfalls kommt der Epilog auf die Liste. Damit anstehende Epiloge so früh wie möglich verarbeitet werden, wird beim Verlassen eines kritischen Abschnittes die Epilogliste bevorzugt abgearbeitet. Epiloge stellen selbst auch kritische Abschnitte dar, demzufolge müssen auch diese mit gesetzter Schloßvariablen bearbeitet werden. Während der Ausführung von Epilogen sollen Unterbrechungen grundsätzlich möglich sein, unabhängig davon, ob der Epilog direkt vom Prolog oder über den Umweg der Epilogliste gestartet wurde.

Graphisch stellt sich der geschilderte Sachverhalt in der Abbildung 2.1 dar.

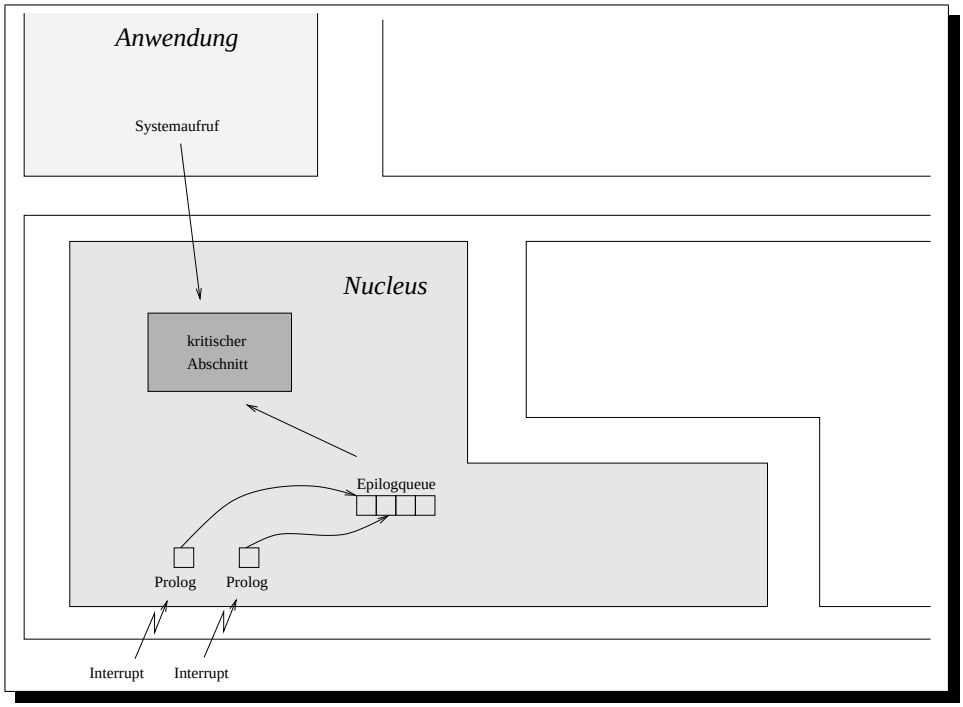

Abbildung 2.1: Standardverhalten in PURE

Ein kleines Problem besteht weiterhin, da die Prologe sowie die Epiloge auf mindestens einer gemeinsamen Datenstruktur arbeiten. Die Epilogliste ist der kritische Datenbestand. Eine Möglichkeit dieser Problematik zu entgehen, bestände in einem kurzzeitigen Unterbinden von Interrupts [3]. Eine viel geeignetere Lösung stellt jedoch die unterbrechungstransparente Listenimplementierung CARGO [10] dar.

[1] First In First Out

Die vorangegangenen Betrachtungen zeigen, daß Unterbrechungen, in der Reihenfolge, in der diese auftraten, verarbeitet werden. In Systemen, in denen Echtzeitanforderungen benötigt werden, ist diese Herangehensweise nicht immer tolerierbar. Ein Fall könnte auftreten, in dem eine Unterbrechung wichtiger ist als eine andere, aber die Reihenfolge in der Epilogliste ermöglicht keine Bevorzugung. Eine ähnliche Situation wird auch bei der Readyliste in der Prozeßverwaltung vorgefunden. Daher werden andere Konzepte vorgeschlagen, die eine bevorzugte Abarbeitung von Objekten gestattet. Dabei soll die Handhabung der Objektvarianten (passiv und aktiv) durch eine Verwaltungsinstanz erbracht werden.

2.2 Entwurf anderer Konzepte

Für die nun im Folgenden vorgestellten Konzepte arbeiten die Listen, in denen Objekte verwaltet werden, nicht mehr nach dem FIFO-Prinzip. Vielmehr werden die Objekte sortiert eingefügt. Die Sortierung erfolgt anhand einer Priorität, die jedes Objekt besitzt.

Zwei Herangehensweisen, wie eine gemeinsame Verwaltung erreicht werden kann, werden beschrieben. Danach erfolgt die Auswahl, welches der Konzepte implementatorisch verwirklicht wurde.

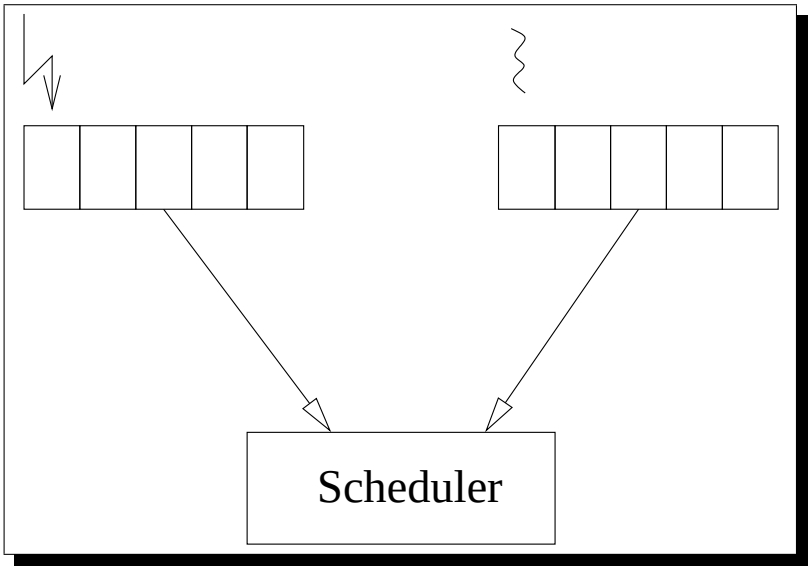

Abbildung 2.2: Verwaltungseinheit mit getrennten Datenstrukturen

2.2.1 Verwaltung durch getrennte Datenstrukturen

Der Abbildung 2.2 ist zu entnehmen, daß der Scheduler zwei Listen verwaltet. Die erste wird für anstehende Unterbrechungen benutzt. Die zweite steht der Prozeßverwaltung zur Verfügung.

Der Scheduler wird beim Aufruf von Systemfunktionen, sowie infolge von Interrupts aktiviert. Dadurch ist gewährleistet, daß die Kontrolle, ob ein Objekt propagiert wird, allein dem Planungswesen obliegt. Die Methode der getrennten Verwaltung birgt auch Nachteile in sich. Zum einen benötigt der Scheduler Wissen über die Objektausprägung, um Objekte in die richtige, dafür vorgesehene, Liste einzusortieren. Zum anderen muß immer auf beiden Listen nachgesehen werden, damit das höherpriorisierte Objekt identifiziert werden kann, um dieses starten zu können. Die Information, welche Objektvariante (passiv oder aktiv) vorliegt, müssen die Objekte selber tragen. Ein erhöhter Speicheraufwand wäre vermutlich die Folge. Durch den Bedarf an zwei Listen und dem daraus folgenden Mehraufwand in der Verwaltung würde der Laufzeitaufwand wahrscheinlich steigen.

Weiterhin ist zu beachten, daß beide Listen die gleiche Strategie verfolgen. Ein nicht Beachten könnte Probleme, wie das der Prioritätsumkehr, bedeuten. Unter Prioritätsumkehr wird ein Ausführen eines Objektes mit niedriger Wertigkeit verstanden, obwohl ein höherwertiges rechenbereit ist. Liegen die Listen nicht sortiert vor oder verfolgen sie unterschiedliche Strategien, tritt ein Reihenfolgeproblem auf. Die Reihenfolge jedoch entscheidet, zu welchem Zeitpunkt Objekte abgearbeitet werden.

2.2.2 Verwaltung mit fusionierten Datenstrukturen

Aus dem vorherigem Abschnitt geht hervor, daß die Listen die gleichen Strategien verfolgen müssen, da sonst mit Prioritätsumkehr zu rechnen ist. Daher ist hier eine Verschmelzung der Listen angedacht. Im Bild 2.3 kann erkannt werden, wie das Planungswesen die Objekte innerhalb einer Datenstruktur arrangiert. Eine Fusion der Listenstrukturen liegt vor. Die Kenntnis über die Objektausprägung ist für das Planungswesen in diesem Konzept unerheblich. Für das Einfügen in die Readyliste braucht keine Zusatzinformation, außer der Priorität, zu existieren. Wenn garantiert ist, daß Objekte ordnungsgemäß in die Liste aufgenommen werden, kann das Problem der Prioritätsumkehr durch die Liste ausgeschlossen werden. Da eine Liste vorliegt, wird nur eine Strategie verfolgt.

Auch wie schon beim ersten Konzept soll der Scheduler immer aktiviert werden, wenn eine Ausnahmesituation auftritt oder ein Systemaufruf stattfand. Demzufolge geht die Kontrolle an das Planungswesen über, wenn ein Objekt nach Ausführung verlangt. Der Scheduler entscheidet anhand der Priorität, ob das Objekt starten soll oder auf die Readyliste verbracht wird.

Das Konzept der fusionierten Datenstrukturen soll das bestehende Schedulingproblem lösen, welches durch die unterschiedlichen Listen für Unterbrechungsbehandlung und Prozeßverwaltung entstand. Dabei war die Prioritätsumkehr das zentrale Problem. Durch die Verwaltung in einer sortierten Liste kann hier Abhilfe geschaffen werden. Hinzu kommt, daß dieses Konzept versucht das Schedulingproblem zu lösen, wobei kein Laufzeitmehraufwand entstehen soll.

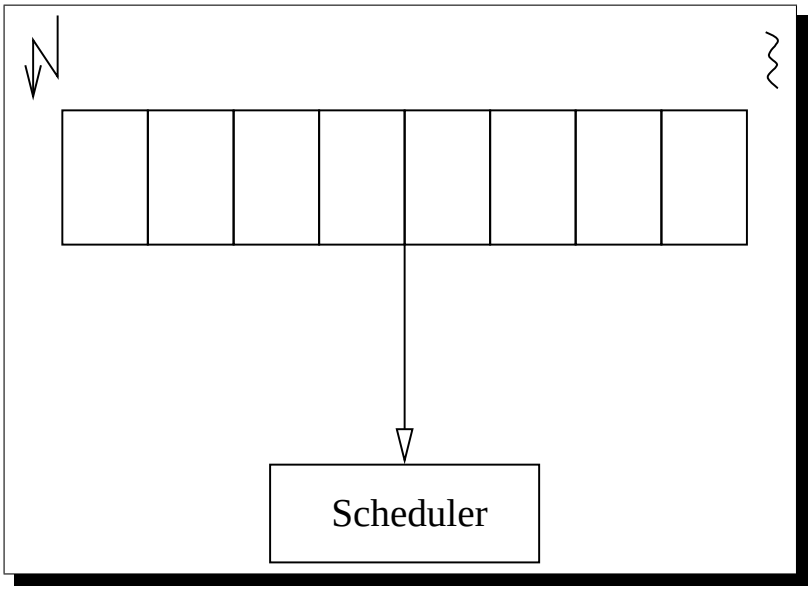

Abbildung 2.3: Verwaltungseinheit mit fusionierten Datenstrukturen

2.2.3 Auswahl des Konzeptes

In dem ersten Konzept wurden zwei Listen benötigt, die nach der gleichen Strategie arbeiteten. Der Scheduler muß demzufolge immer beide bearbeiten. Weiterhin muß das Planungswesen Kenntnis über die Objektausprägung besitzen. Diese Nachteile gibt es beim zweiten Ansatz nicht. Aus diesen Überlegung wurde der Entschluß gefaßt, den zweiten Ansatz, mit den fusionierten Datenstrukturen, in der vorliegenden Arbeit implementatorisch zu verwirklichen.

Ein weiterer Grund für die Wahl war, daß bei dem zweiten Konzept von einem geringeren Synchronisationsaufwand ausgegangen wurde. Dies ist nicht unerheblich für die Komplexität und Überschaubarkeit der Implementierung. Eine „leichtere“ Verifizierbarkeit muß angestrebt werden.

Weiterhin wurde davon ausgegangen, daß durch diese Implementation die Speichergrößen und der Laufzeitaufwand geringer wären, als bei dem ersten Ansatz. Um jedoch einen Vergleich zu ermöglichen, müßte auch die andere Version verwirklicht werden.

Kapitel 3

Entwurf und Implementierung

In den folgenden Abschnitten wird beschrieben wie priorisierte Objekte, passive sowie aktive, aufgebaut sind. Desweiteren wird auf die gemeinsame Listenverwaltung und das präemtive prioritätsorientierte Planungswesen eingegangen. Hierbei werden der Aufbau und die Funktionsweise erklärt.

3.1 Priorisierte Objekte

3.1.1 Entwurf

In der Betriebssystemumgebung, insbesondere in echtzeitkritischen Anwendungen, existieren Objekte, die bevorzugt behandelt werden müssen. Weiterhin muß sichergestellt werden, daß Objekte bis zu einem vorgegebenen Zeitpunkt terminieren. Durch eine hohe Priorisierung kann sichergestellt werden, daß die Objekte so wenig wie möglich in ihrer Arbeit unterbrochen werden. Prinzipiell können Prioritäten auf unterschiedliche Weise verwaltet werden. Eine Möglichkeit besteht in dem Errechnen der Priorität durch eine Funktion. Dabei wird beispielsweise die Wertigkeit eines Objektes um so größer, je länger es auf die CPU wartet. Bei Echtzeitanwendungen ist die Deadline das entscheidende Kriterium, wobei diese für die Prioritätsberechnug herangezogen werden würde.

Für statische Prioritäten brauchen keine Berechnungen durchgeführt werden. Daher ist eine weitere Methode, Prioritäten zu verwalten, das Speichern in einer objektlokalen Variablen. Das Speichern schließt natürlich nicht das dynamische Verwalten aus, denn die Variable kann gelesen und geschrieben werden.

In dem System, welches hier zur Diskusion steht, werden nur statische Prioritäten verwendet. Daher ist die Wertigkeit des Objektes in einer lokalen Variablen untergebracht.

Das priorisierte Objekt dient als Grundbaustein der entstehenden Klassenhierarchie. Der Name der Basisklasse ist *Valence*[1]. Wie in der Abbildung 3.1 zu erkennen ist, existieren einige Methoden, um auf die Wertigkeit Einfluß zu nehmen. Im Folgenden wird kurz erläutert, welche Aufgaben die einzelnen Methoden besitzen.

[1] valence(engl.) = Wertigkeit

const Valence& Valence::value() const
Mit `value()` wird eine konstante Referenz auf die Valence des Objektes geliefert.

Valence Valence::value(const Valence&)
Die Methode dient dazu einen neuen Wert zu setzen und gleichzeitig den Alten zurück zu liefern.

int Valence::operator > (const Valence&) const
Dieser Operator gestattet es zwei *Valence*-Objekte mit einander zu vergleichen.

```
typedef unsigned long type_t;

class Valence {
      type_t level;
public:
      Valence();
      Valence(type_t);
      Valence(const Valence&);

      Valence& operator = (const Valence&);
      Valence& operator = (const type_t&);

      const Valence& value() const;
      Valence value(const Valence&);

      int operator > (const Valence&) const;
};
```

Abbildung 3.1: Valence-Schnittstelle

Dem Operator > kommt, in den weiterführenden Betrachtungen, eine bedeutende Rolle zu. Dieser wird benötigt, um zum Beispiel Objekte in der richtigen Reihenfolge in die Readyliste zu integrieren und weiterhin zur Entscheidungsfindung für die Propagation von Objekten. Für das Anheben und Senken der Priorität stehen die Lese- und Setzmethoden zur Verfügung. Warum Prioritäten verändert werden müssen wird in dem Abschnitt 3.3 über Scheduler behandelt.

3.1.2 Konfiguration

Da die maximal benötigte Anzahl an Prioritäten von System zu System unterschiedlich ist, besteht die Möglichkeit einer Konfiguration. Die Abbildung 3.1 zeigt eine Einstellung für ein System, welches 2^{32} verschiedener Wertigkeiten bedarf. Zu sehen ist der Typ *type_t*. Dieser kann auf verschiedene Weise ausgelegt sein. So könnte z.B. der Typ *type_t* auch durch ein *unsigned char* gekennzeichnet sein, wobei 2^8 der Prioritätsanzahl entsprechen würde. Ist diese Anzahl

für das System ausreichend, kann Speicherplatz gegenüber der voreingestellten Variante eingespart werden.

Wenn die eingebauten Typen der Sprache C++ nicht ausreichend sind, kann hier auch ein benutzerdefinierter Typ verwendet werden. Selbst das dynamische Verwalten, mit Hilfe von Funktionen, ist durch diesen Konfigurationsmechanismus gewährleistet.

3.2 Die Listenverwaltung

3.2.1 Entwurf

Wie aus den vorangegangenen Betrachtungen hervorgegangen, werden in der Liste sowohl passive als auch aktive Objekte zu verwalten sein. Die Objekttausprägung ist für die Liste jedoch nicht relevant. Einzig die Priorität der Objekte spielt eine Rolle. Objekte die in der Liste gehalten werden, sind vom Typ Usher[2]. Wie der Abbildung 3.2 zu entnehmen ist, wurde die Klasse Usher von Chain und Valence abgeleitet. Die Basisklasse *Chain*[3] stellt die Funktionen und Fähigkeiten zur Verkettung bereit. Wie aus dem vorherigem Abschnitt bekannt, stellt die *Valence* die Priorität zur Verfügung. Durch das Zusammenerben der beiden Basistypen ergibt sich ein verkettungsfähiges priorisiertes Objekt. Für das Verwalten sind die Fähigkeiten und der Informationsgehalt somit ausreichend. Erst zu einem späteren Zeitpunkt werden die zusätzlichen Bestandteile der passiven und aktiven Objekte hinzugefügt.

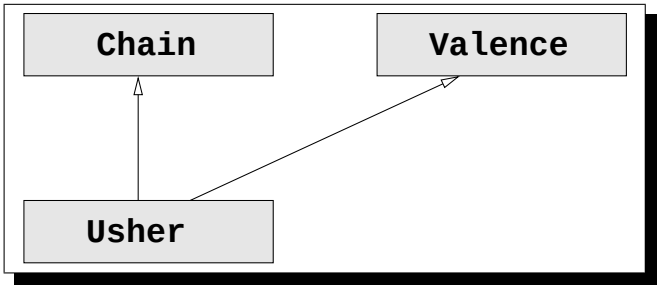

Abbildung 3.2: Vererbungshierarchie der Listenverwaltung

3.2.2 Bestimmen des Funktionsumfanges

Es stellt sich natürlich die Frage, welcher Funktionsumfang benötigt wird. Zu nennen ist folgender

1. einsortieren eines Elementes
2. das erste Element entfernen
3. ein beliebiges Element herausnehmen.

In dem Bild 3.3 sind die Methoden zu erkennen. Die Funktion defer()[4] gewährleistet das ordnungsgemäße Einsortieren von Objekten. Ausgehend von der Priorität wird ein Objekt innerhalb der Liste plaziert. Die Prioritätenliste ist beginnend mit dem Element mit der größten Wertigkeit absteigend sortiert.

Die Methode head()[5] liefert den Kopf der Liste. Dieser zeigt auf das erste Element, folglich dem am höchstpriorisiertem Objekt.

[2] usher(engl.) = Platzanweiser
[3] chain(engl.) = Kette
[4] to defer(engl.) = aufschieben, verzögern
[5] head(engl.) = Kopf

```
class Usher: public Chain, public Valence {
      static Chain* Head;
public:
      Usher();
      Usher(const Valence&);
      Usher(const Usher&);
      Usher& operator = (const Usher&);
      Chain* head() const;
      void defer();
      Usher* fetch();
      void remove();
};
```

Abbildung 3.3: Usher-Schnittstelle

Desweiteren kann die Funktion remove()[6] in der Schnittstellendefinition betrachtet werden. Diese dient dem Entfernen des aufrufenden Objektes aus der Liste. Dafür muß das Element zuerst in der Liste gesucht werden, indem die Liste sukzessiv von vorn nach hinten durchlaufen wird.

Die Funktion fetch()[7] liefert das erste Element der Liste und löscht es von dieser. Sie wurde zusätzlich zur remove()-Methode entwickelt, da beim Herausnehmen des ersten Elementes die Liste nicht durchsucht werden muß. Somit weist sie einen geringeren Laufzeitaufwand auf, welches zum Beispiel Echtzeitanforderungen entgegen kommt.

Der Initialzustand der Liste ist, wie der Abbildung 3.4 zu entnehmen ist, nicht leer. Der *Head*-Pointer zeigt, wenn noch kein ausführbares Objekt in der Liste steht, auf ein Element mit der Priorität null. Der Grund für dieses Vorgehen besteht in der möglichst effizienten Abwicklung der Behandlung der Objekte. Da die Priorität null durch kein weiteres Objekt angenommen werden darf, braucht nie überprüft werden, ob die Liste eventuell leer ist. Dadurch brauchen diverse Vergleiche innerhalb der Listenimplementierung nicht durchgeführt werden.

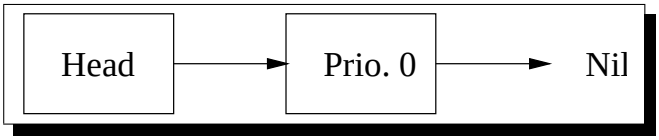

Abbildung 3.4: Initialzustand der Liste

[6] to remove(engl.) = entfernen
[7] to fetch(engl.) = abrufen, holen

3.2.3 Problem der Nebenläufigkeit

Da in dem System sowohl Unterbrechungen als auch Systemdienste an den gleichen Datenstrukturen arbeiten, kann es zu Inkonsistenzen kommen. Daher ist es von Bedeutung die Bereiche aufzuspüren, in denen Operationen auf den Datenstrukturen kritisch sind. Dadurch wird eine aufwendigere Listenimplementierung notwendig.

Die Methoden des *Usher* können sich beliebig unterbrechen. Folgende Fälle können eintreten:

1. `defer()` unterbricht `defer()`,
2. `defer()` unterbricht `fetch()`,
3. `defer()` unterbricht `remove()`,
4. `fetch()` unterbricht `defer()`,
5. `fetch()` unterbricht `remove()`,
6. `remove()` unterbricht `defer()` und
7. `remove()` unterbricht `remove()`.

Die beiden Fälle

8. `fetch()` unterbricht `fetch()`,
9. `remove()` unterbricht `fetch()`,

können nicht eintreten, da diese durch die Sicherungsmaßnahmen im Scheduler ausgeschlossen wurden.

Die Unterbrechungssituationen der Punkte vier bis sieben bedingen keiner besonderen Behandlung, da diese per Definition auf unterschiedlichen Teilabschnitten der Liste operieren. Garantiert wird dies durch höhere Softwareschichten. Somit stellen nur die ersten drei Unterbrechungszenarien eine mögliche Konfliktsituation dar, wenn nebenläufig an den gleichen Daten gearbeitet wird.

Nun gibt es verschiedenste Möglichkeiten der Synchronisation für kritische Abschnitte. Einige dieser sollen kurz angezeigt und erläutert werden.

- Unterdrücken von Unterbrechungen
- Schloßvariable
- Semaphore
- Optimistische Verfahren
  - Compare And Swap
  - Load Linked Store Conditional

Im Folgenden wird auf die Punkte näher eingegangen.

3.2.3.1 Unterdrücken von Unterbrechungen

Um während der Ausführung von kritischen Abschnitten nicht durch Interrupts unterbrochen zu werden, gibt es bei vielen Prozessoren die Möglichkeit diese hardwareseitig zu unterbinden. Dafür stellen die verschiedenen Prozessoren Maschineninstruktionen bereit, die das Setzen oder das Maskieren von Interruptleveln erlauben. Im allgemeinen sind diese Anweisungen jedoch nur im priviligierten Modus ausführbar. Daher können Anwendungen von diesen kaum profitieren, außer das Betriebssystem bietet eine Schnittstelle für deren Benutzung.

In Einprozessorsystemen ist das Unterdrücken von Interrupt eine gute Möglichkeit kritische Abschnitte zu schützen, jedoch ist es keine ideale Lösung, weil Unterbrechungen verloren gehen können. Dadurch kann es vorkommen, daß auf Ereignisse überhaupt nicht reagiert wird. Dadurch ist dieses bei Echtzeitsystemen nur mit Bedacht einzusetzen.

3.2.3.2 Schloßvariable

Für den gegenseitigen Ausschluß wurde auch die Methode der Schloßvariable erdacht. Hierbei wird um eine globale Datenstruktur zu schützen eine globale Variable genutzt. Der Zustand der Variablen gibt Aufschluß, ob der kritische Bereich frei oder belegt ist. Möchte z.B. eine Anwendung einen kritischen Abschnitt betretten so muß diese eine Methode aufrufen, die die Schloßvariable als belegt kennzeichnet. Beim Verlassen wird ebenfalls eine Funktion aufgerufen, die die Variable so modifiziert, so daß der Abschnitt als frei gekennzeichnet wird.

Nun ergibt sich hier aber das selbe Problem wofür die Schloßvariable die Lösung darstellen soll. Denn zwischen dem Abfragen der Variable, sowie dem Setzen kann eine Unterbrechung stattfinden. Eine Lösung auf Einprozessorsystemen wäre das Unterdrücken von Interrupts. Der kritische Abschnitt des Vergleichens und Setzens der Variablen ist jedoch viel kürzer, als der, der während eines kritischen Abschnittes, bei dem globale Datenstrukturen modifiziert werden. Somit entsteht ein reaktionfreudigeres System. Das Zurücksetzen stellt keinen kritischen Bereich dar und muß demzufolge auch nicht geschützt werden.

Ein Nachteil der Methode stellt das aktive Warten der Prozesse dar, wenn die Schloßvariable belegt ist. Dadurch ist sie für die Synchronisation zwischen Betriebsysytemaufrufen und Interrupts unbrauchbar. Wenn z.B. ein Prozess einen kritischen Abschnitt betreten hat, der dann durch einen Interrupt auch betreten werden soll, muß dieser aktiv auf das Freiwerden warten. Jedoch kann nur der Prozess, auf dessen Kontext die Unterbrechung läuft, diesen wieder freigeben, wozu er nie die Gelegenheit erhalten wird. Wie zu sehen ist, ist diese Synchronisationationsform nicht für den benötigten Fall geeignet.

3.2.3.3 Semaphore

Ein entscheidender Nachteil der Schloßvariablen ist, daß auf das Freiwerden aktiv gewartet wird. Eine schlechte Systemauslastung oder gar ein Deadlock[8] des Systems kann die Folge sein. Bei der Semaphore wird ähnlich wie bei den Schloßvariablen der Zustand in einer globalen Variable vermerkt. Will ein Prozess den kritischen Abschnitt betreten, so muß er die Methode `wait()` aufrufen. Ist der Bereich frei, so kann er ohne Hindernisse fortfahren. Ist hingegen der Abschnitt belegt, wird der Prozess blockiert. Wenn nun ein Prozess den kritischen Bereich durch `signal()` verläßt, so wird der blockierte Prozess wieder deblockiert.

Für den gegenseitigen Ausschluß von Unterbrechungsbehandlungen und Systemfunktionen sind Semaphore jedoch nicht geeignet. Wie schon bei den Schloßvariablen beschrieben, würde es auch hier zu Verklemmungen kommen, da der Interrupt blockieren würde, und der Prozess, der in der Lage wäre ihn zu deblockieren, wird durch ihn selber überdeckt.

3.2.3.4 Optimistische Verfahren

In den optimistischen Verfahren wird versucht Operationen abzuschliessen und hernach zu prüfen, ob diese erfolgreich verlaufen sind.

Soll eine Variable in einem System hochgezählt werden, gestaltet sich das auf Maschinen ohne eine hardwareseitig Unterstützung recht schwierig. Denn einen nebenläufigen Zugriff auf diese Speicherstelle zu erkennen, ist problematisch. Aus diesem Grund bieten einige Prozessoren Hardwareinstruktionen an. In einigen Familiemitgliedern der CISC-Rechner ist das *C*ompare *A*nd *S*wap anzutreffen. Bei der RISC-Architektur wird in einigen Prozessoren das *L*oad *L*inked *S*tore *C*onditional angeboten. Die beiden folgenden Abschnitte behandeln wie dieses funktionieren und anzuwenden sind.

3.2.3.4.1 *C*ompare *A*nd *S*wap

Wenn eine Speicherstelle verändert werden soll, so muß sichergestellt werden, daß keine nebenläufigen Zugriffe stattfinden. Wenn der Zugriff und die einhergehende Veränderung der Variable abhänig vom Inhalt gemacht werden muß, so erfolgt im voraus das Speichern des alten Wertes.

Das `CAS` ermöglicht den atomaren Vergleich der Speicherstelle mit einem zu übergebenen Wert(alten Wert), wenn dieser Vergleich als Ergebnis die Gleichheit liefert, so kann ein zweiter Wert(neuer Wert) auf diese Speicherstelle geschrieben werden. Als Ergebnis liefert das `CAS` einen Zustand, der Auskunft über Erfolg oder Mißerfolg des Befehls gibt.

In der Abbildung 3.5 ist die prinzipielle Funktionsweise zu erkennen. Die Aufrufe *ExclusiveEnter*, sowie *ExclusiveLeave* werden durch die Maschineninstruktion durchgeführt.

Bietet der Prozessor keine derartige Anweisung an, so kann diese wie in der Abbildung 3.5 zu sehen ist auch in Software nachgebaut werden. Dabei

[8] deadlock(engl.) = Blockierung

```
1:   bool CAS( void* addr, size_t old_value, size_t new_value){
2:        <ExclusiveEnter>
3:        bool result=( *addr == old_value);
4:        if (result)
5:            *addr=new_value;
6:        <ExclusiveLeave>
7:        return result;
8:   }
```

Abbildung 3.5: Compare And Swap

muß das *ExclusiveEnter*, sowie *ExclusiveLeave* durch Speren und Freigeben von Interrupts implementiert werden. In den Abbildungen 3.6 sowie 3.7 ist dieses zu erkennen.

Diese Methode kann für die Synchronisation von Listen, die durch Interrupts und Systemfunktionen behandelt werden, angewendet werden.

```
1:   void ExclusivEnter(){
2:        disable_int;       // Interrupts abschalten
3:        lock;              // Bus locken
4:   }
```

Abbildung 3.6: Implementierung von ExclusivEnter

```
1:   void ExclusivLeave(){
2:        unlock;            // Bus freigeben
3:        enable_int;        // Interrupts abschalten
4:   }
```

Abbildung 3.7: Implementierung von ExclusivLeave

3.2.3.4.2 *L*oad *L*inked *S*tore *C*onditional

Hier wird ähnlich dem CAS vorgegangen. Da aber auf RISC-Maschinen die Ausführung von Instruktionen immer innerhalb eines Taktes erfolgt, ist das CAS in zwei Funktionsblöcke aufgeteilt.

Im erste Block Load Linked (kurz LL) erfolgt die Reservierung einer Speicherstelle. Dadurch wird angezeigt, daß darauf ein atomarer Speicherzugriff erfolgen

soll. Bei dem Store Conditional (kurz SC) wird geprüft, ob die Reservierung noch besteht, ist dies der Fall, erfolgt das Schreiben eines neuen Wertes.

Auch hiermit kann eine Synchronisation zwischen Unterbrechungen und Systemfunktionen vorgenommen werden.

3.2.3.5 Wahl des Synchronisationsmechnismuses

In den vorangegangenen Abschnitten wurden die einzelnen Synchronisationsmethoden beschrieben. Dabei wurde auf die Funktionsweise und Einsatzgebiete jeder einzelnen eingegangen. In der Tabelle 3.1 ist noch einmal übersichtlich zusammengestellt, welche Unterbrechungsszenarien die einzelnen Schutzmöglichkeiten abdeckten. Dabei sind folgende Szenarien vorstellbar:

1. Anwendung unterbricht Anwendung,
2. Interrupt unterbricht Anwendung und
3. Interrupt unterbricht Interrupt.

Da eine Anwendung nie einen Interrupt unterbrechen kann, ist dieser Fall ausgeklammert.

	Anw / Anw	Anw / Int	Int / Int
Semaphore	x	-	-
Schloßvariable	x	-	-
enable/diable Int	x	x	x
prozessorabhänig			
CAS	x	x	x
LLSC	x	x	x

Tabelle 3.1: Möglichkeiten der Synchronisation

Unter Anwendungen soll hier nicht nur die Anwendung als solches verstanden werden, sondern auch ein Systemaufruf, der von einer Anwendung abgesetz wurde. Folglich ist die Synchronisation zwischen dem Systemaufruf und dem Interrupt von Bedeutung, sobald diese auf den gleichen Datenstrukturen arbeiten.

Auch die Synchronisation von Interrupts untereinander ist in dem entstehenden System wichtig, da Unterbrechungen zu jedem Zeitpunkt auftreten können. Wenn beispielsweise ein Interrupt niederer Priorität abgearbeitet wird, kann er durch einen höherpriviligierten verdrängt werden. Dadurch besteht in jenen Situation Synchronisationsbedarf.

Aus den geschilderten Gründen scheiden die Semaphore und die Schloßvariable aus, da sie nicht für die Synchronisation von Anwendungen und Unterbrechungen verwendet werden können.

Aus der Tabelle geht hervor, daß die prozessorabhänigen Instruktionen und das Sperren und Freigeben von Interrupts als Lösungsvariante übrig bleibt.

Wenn Interrupts abgeschaltet werden, können in der Zeit, während Unterbrechungen untersagt sind, einige verloren gehen. In System mit Echtzeitanforderungen ist das nicht tolerierbar. Systeme die reaktiosfreudig aufgebaut sein sollen, müssen so wenig als möglich die Unterbrechungen abgeschaltet haben. Da hier ein System entstehen soll, welches möglichst reaktionsfreudig ausgelegt ist, wird auf das Abschalten von Unterbrechungen verzichtet. Stattdessen werden die prozessorabhänigen Anweisungen benutzt. Hier wurde das *CAS* gewählt, da es auf dem Entwicklungssystem als Maschinenanweisung zur Verfügung stand. Wie oben gezeigt, kann das *CAS* aber auch leicht durch Software nachgebildet werden.

3.2.4 Einfügen eines Listenelementes

Um ein Element an der richtigen Position innerhalb der Liste zu positionieren, müssen einige Randbedingungen erfüllt werden. Folgender Ablaufplan wird während dem Einsortieren durchgespielt.

1. lesen des Listenkopfes
2. ausgehend vom Kopf der Liste das Element suchen, welches eine kleinere Priorität aufweist als das einzufügende Objekt
3. das Objekt als Nachfolger des eigenen Elementes vermerken
4. das eigene Element als Nachfolger des Vorgängers eintragen

Wie schon beschrieben, arbeiten sowohl Unterbrechungen, als auch Systemfunktionen an den gleichen Datenstrukturen. Hier ist die gemeinsame Liste für passive und aktive Objekte von Bedeutung.

Das Einfügen von Elementen kann überlappt ausgeführt werden. Daher müssen die Bereiche innerhalb der Methode identifiziert werden, in denen die Nebenläufigkeit ein Problem darstellt. Sind die Abschnitte aufgespürt worden, kann mit der Behandlung der kritischen Sequenzen begonnen werden.

Das Lesen des Listenkopfes stellt keinen kritische Bereiche dar. Selbst wenn eine Veränderung stattfinden würde, wird es in der folgenden Abarbeitung erkannt. Das Durchsuchen der Liste und das Auffinden der korrekten Position stellt ebenso wenig eine verhängnisvolle Situation dar, weil keinerlei Manipulationen an der Liste vorgenommen werden. Wenn die richtige Einfügestelle gefunden ist, erfolgt das Vermerken des eigenen Nachfolgers. Wiederum nicht problematisch, da nur eigenen Objektdaten verändert werden, zu denen momentan keine anderen Funktionen Zugang erlangen. Der nächste Schritt das Einhängen in die Liste selber stellt auch kein Problem, bezüglich Unterbrechen dar. Ein Problem entsteht sobald eine Unterbrechung zwischen dem dritten und dem vierten Arbeitsschritt erfolgt, wobei ein Element an genau der gleichen Position eingefügt werden soll. Ein Element, welches an einer anderen Position plaziert werden muß, stellt keine Gefahr da, weil die Einfügeoperationen auf unterschiedlichen Teilen der Liste arbeiten und somit mit unterschiedlichen Daten.

Was würde geschehen, wenn keine Behandlung des Fehlerfalls unternommen würde?

Die folgenden drei Abbildungen beschreiben die entstehende Situation. Die Abbildung 3.8 zeigt den Zustand nachdem der Nachfolger in das Element eingefügt worden ist. Hernach erfolgt eine Unterbrechung, welche ihrerseits eine Einfügeoperation absetzt. Da die Liste zu diesem Zeitpunkt immer noch „leer“ ist, erfolgt das Einfügen des Interruptobjektes an der selben Position. Somit ergibt sich der kritischer Fall. Die Abbildung 3.9 zeigt die Lage, wie sie nach dem beenden der Interruptbehandlung vorgefunden wird. Da die erste Einfügeopera-

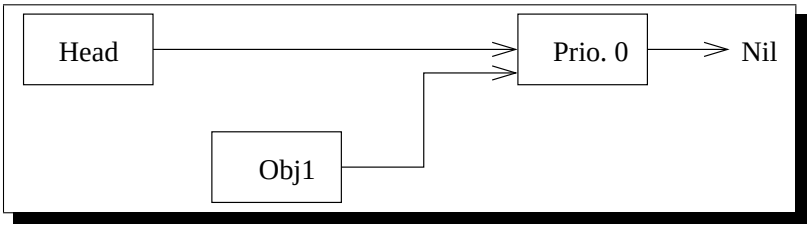

Abbildung 3.8: Unvollendeter Einfügevorgang

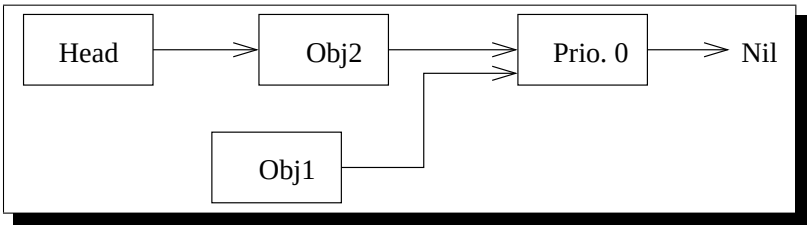

Abbildung 3.9: Einfügen eines Objektes während eines unterbrochenen Einfügevorgang

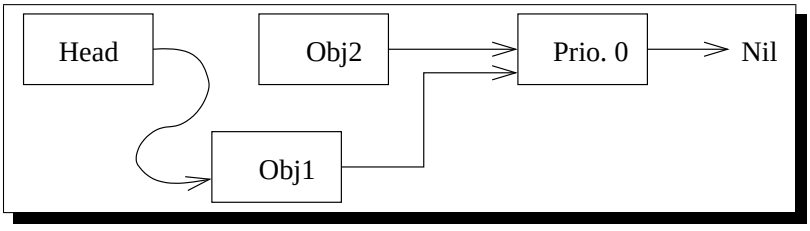

Abbildung 3.10: Beenden des unterbrochenen Einfügevorganges

tion keine Kenntnis über die Unterbrechung hat, kann auch nicht entsprechend auf diese reagiert werden. Folglich wird hier noch mit alten Werten gearbeitet. Dadurch erfolgt ein Einketten am Kopf der Liste, wodurch das Interruptobjekt aus der Liste verschwindet. In Bild 3.10 ist das noch einmal grafisch dargestellt.

Wie zu sehen ist, bedarf diese kritische Situation einer Behandlung, die die Konsistenz der Liste wart.

Aus dem vorherigen Abschnitt 3.2.3.5 geht hervor, daß zur Synchronisation von Unterbrechungen und Systemaufrufen die optimistischen Verfahren benutzt werden. Dabei kommt die vom Prozessor unterstützte Funktion *CAS* zum Einsatz. Mit Hilfe dieser kann eine nebenläufige Veränderung einer Speicherstelle erkannt werden.

Prinzipiell ist die Situation, in der der Fehler auftreten kann, die gleiche. Somit ist wiederum eine Unterbrechung zwischen dem dritten und dem vierten Arbeitsschritt kritisch. Wird hier jedoch beim vierten Punkt ein *CAS* verwendet, so kann die überlappte Abarbeitung erkannt werden. Ist der Fehlerfall eingetreten, erfolgt eine Wiederholung des Einfügevorganges. Somit ändert sich die Ablaufsequenz in:

1. lesen des Kopfes
2. Einfügeposition suchen
3. Nachfolger eintragen
4. *CAS* aufrufen und Element einketten
5. wenn *CAS* fehlgeschlagen, wiederhole ab Punkt 1.

Wenn das obere Beispiel ein weiteres mal betrachtet wird, ergibt sich nach der in Abbildung 3.10 gezeigten Situation eine Erkennung des Fehlerfalles. Dadurch erfolgt eine erneute Abarbeitung der Sequenz und das Element wird korrekt eingefügt. Das Bild 3.11 zeigt den jetzt richtigen Endzustand.

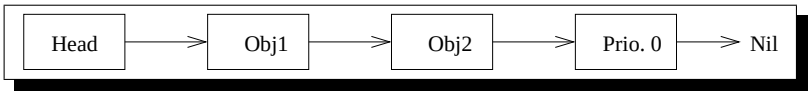

Abbildung 3.11: korrektes Beenden des unterbrochenen Einfügevorganges

3.2.4.1 Optimierung des Einfügevorganges

Die Routine, wie sie oben beschrieben wurde, muß nach jedem nebenläufigen Unterbrechen, wodurch infolge das *CAS* den Fehlerfall zeigt, erneut die Einfügeposition suchen. Bei einer hinreichend langen Liste wäre eine erhöhte Laufzeit das Ergebnis. Um jedoch die Anwendungen nicht mehr als nötig warten zu lassen, wurde eine Optimierung des Algorithmuses vorgenommen.

Grundsätzlich braucht nicht vom Anfang der Liste neu begonnen zu werden. Wenn ein oder mehrere Elemente an die gleiche Position eingekettet wurden, muß überprüft werden, ob die Bedingung der richtigen Reihenfolge noch garantiert ist. Somit wird ab der Einfügeposition mit dem Suchen begonnen. Die Sequenz wurde in folgende geändert.

1. lesen des Listenkopf und vermerken als Einfügeposition

2. ab vermerkter Einfügeposition suchen
3. vermerken der neuen Einfügeposition
4. Nachfolger eintragen
5. *CAS* aufrufen und Element einketten
6. wenn *CAS* fehlgeschlagen, wiederhole ab Punkt 2.

Durch diese Maßnahmen wird eine Laufzeitverbesserung erreicht, wenn tatsächlich eine Unterbrechung ein Objekt an genau der selben Stelle einsetzen wollte.

3.2.4.1.1 Realisierungmöglichkeiten der Optimierung

Realisiert wurde der Algorithmus der in Abschnitt 3.2.4.1 beschrieben wurde. Das vermerken der Einfügeposition kann über zwei verschiedene Wege erreicht werden.

Die erste Variante, der nachlaufende Zeiger, beschreibt ein Verfahren, bei dem ein Zeiger immer eine Position zurückhängt.

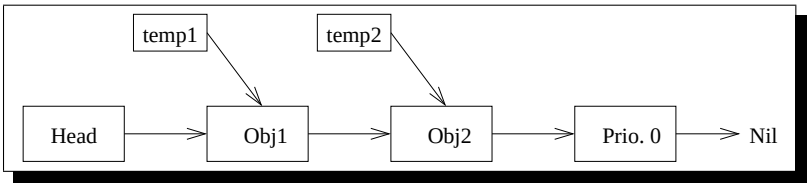

Abbildung 3.12: Realisierung mit der zwei Zeiger Technik

In der Abbildung 3.12 sind die beiden Zeiger *temp1* und *temp2* zu sehen. Der Pointer *temp2* wird zum Vergleichen und Auffinden der Position benötigt. Damit das Einhängen in die Kette gelingt, muß der Vorgänger bekannt sein. Für diese Aufgabe ist der *temp1* gedacht. Bei einem Weitersetzen in der Liste müssen demzufolge immer beide Pointer aktualisiert werden.

Die zweite Lösungsidee realisiert die gestellte Aufgabe durch einen kleinen Trick. Diese Technik nutzt die vorhandenen Pointer, welche bereits den Verkettungen in der Liste dienen. Dies bedeutet, es wird immer ein Verkettunszeiger vorausgeschaut, um die richtige Einfügestelle zu finden. In Bild 3.13 ist der beschriebenen Sachverhalt verdeutlicht. Die Zeiger mit den Nummern eins und zwei sind identisch. Sie stellen genau den selben Pointer dar. Diese Darstellung soll nur das Durchgreifen auf die vorhandenen Pointer der Liste symbolisieren.

Bei beiden Methoden ist die Manipulation des Kopfes der Liste ein Problem, da es keinen Vorgänger gibt. Es müßt demzufolge eine zusätzliche Abfrage benutzt werden, um zu erkennen, ob der Listenkopf einer Veränderung bedarf. Umgehen läßt sich diese Situation, indem der Zeiger, welcher allgemein zur Anpassung benutzt wird nicht anfangs auf den Listenkopf direkt zeigt, sondern die Adresse des selben beinhaltet. Dadurch kann dieser direkt modifiziert werden.

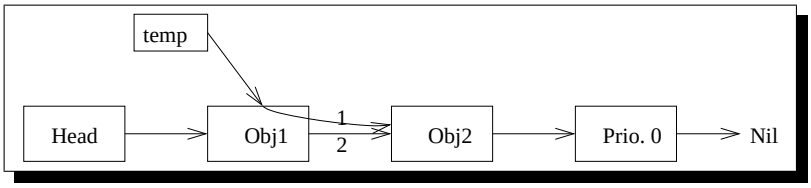

Abbildung 3.13: Realisierung mit der Zeiger auf Zeiger Technik

Für die technische Realisierung wurde der zweite Ansatz gewählt. Bei der zweiten Methodik reduziert sich der Laufzeitaufwand, weil ein Pointer, pro Suchdurchgang, weniger aktualisiert werden muß. Außerdem werden Speicherresourcen gespart, da kein zusätzliche Pointer gebraucht wird.

3.2.4.1.2 Implementationsdetails

Die Methode wie sie im Bild 3.14 steht, entspricht dem Algorithmus wie er beschrieben wurde. Dabei wurde „die Zeiger zeigt auf Zeiger“ Technik angewendet. Wie zu erkennen ist, findet keine Überprüfung auf das Listenende statt, da per Definition immer ein Element mit der Priorität null am Ende der Liste hängt und diese Priorität kein anderes Objekt annehmen darf. Ferner ist in Zeile zwei der Abbilung gezeigt, wie die Initalisierung des Such- und Einfügepointer vorgenommen wurde.

```
1:    void Usher::defer(){
2:         register Chain **temp=&Head;
3:         do{
4:              while (**(Usher **)temp > *this) {
5:                   temp=(Chain **)*temp;
6:              }
7:         select(*temp);
8:         } while ( !CAS((Chain *)temp, select(), this) );
9:    }
```

Abbildung 3.14: Die Methode `defer()`

Die *while*-Schleife in den Zeilen vier bis sechs sucht die Einfügeposition in der Liste, wobei die Codezeile fünf das Fortschreiten realisiert. Dabei wird dem *temp*-Zeiger, bei jedem Schritt, die Adresse des nächsten Elementes zugewiesen. Mit der Methode `select(argument)`, welche die Basisklasse *Chain* bereitstellt, wird der Nachfolger des Elementes eingetragen. Die gleiche Methode ohne Argument liefert den Nachfolger. Somit wird durch `select(*temp)` der Nachfolger des Elementes eingetragen. Abschließend wird in der Zeile acht das Einketten erbracht. Schlägt dieses fehl, da eine nebenläufige Aktion ein Element an der selben Position eintrug, so wird ab Zeile drei wiederholt. Ist das *CAS* erfolgreich,

so wurde das Objekt korrekt in der Liste angeordnet.

3.2.5 Entfernen eines beliebigen Listenelementes

Die Methode `remove()`, die Gegenstand dieses Abschnittes ist, wird dazu genutzt Objekte aus der Liste auszuketten. Wie der Aufrufsyntax zu entnehmen ist, erhält die Funktion kein Argument. Folglich muß ein Objekt, welches entfernt werden soll, auf sich selber den `remove()`-Aufruf ausführen. Der Algorithmus nach dem der Entnahmevorgang abläuft ist formal folgender:

1. Lesen des Listenkopfes
2. Suchen des Vorgängers des zu entfernenden Elementes
3. In den Vorgänger den Objektnachfolger einhängen

Durch diesen simplen Ablaufplan wird das Element aus der Liste entfernt. Das Problem hierbei sind jedoch die schon beschriebenen asynchronen Unterbrechungen. Aus dem Unterabschnitt 3.2.3 dieses Kapitels geht hervor, daß die Funktion durch eine oder mehrere Einfügeoperationen und auch durch Entnahmevorgänge unterbrochen werden kann. Die kritischen Situationen können aber nur durch Einfügevorgänge eintreten, da per Definition Entnahmeoperationen immer auf unterschiedlichen Listenteilen arbeiten und somit auf unterschiedlichen Datenstrukturen. Bleibt der Fall des Aufrufes einer Einfügefunktion. Wird der obere Fahrplan betrachtet, ist jeder einzelne Arbeitsschritt für sich genommen nicht kritisch, aber die Sequenz. Ins Auge fallen sofort die Positionen zwei und drei. Erfolgt zwischen diesen beiden Anweisungen eine Unterbrechung, die ein Objekt an den Vorgänger anhängt, ändert sich der Vorgänger des zu entnehmenden Objektes.

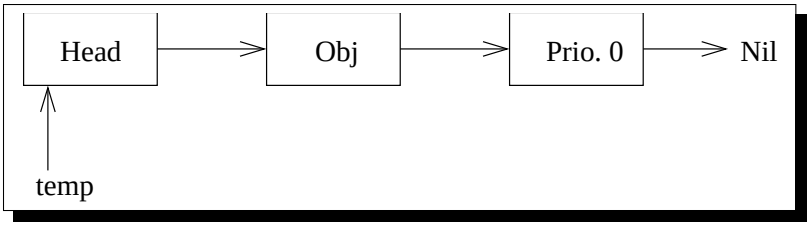

Abbildung 3.15: Ausgangssituation vor dem Ausketten

Das Bild 3.15 zeigt die Situation nach dem zweiten Arbeitsschritt. Der Zeiger *temp* verweist auf den Vorgänger. Nun erfolgt eine Unterbrechung die ein Objekt, gemäß der Priorität, genau an den Vorgänger anfügt. Veranschaulicht stellt sich der Sachverhalt in Abbildung 3.16 dar. Hierdurch wurde der Vorgänger des Elementes nebenläufig geändert. Wird weiterhin, nach dem Interrupt, der Algorithmus abgearbeitet, entsteht die Situation, die in dem Bild 3.17 grafisch dargestellt ist.

Das bedeutet ein sofortiges Ausketten des durch den Interrupt eingefügten Objektes. Das ist jedoch ein Fehler. Ein Erkennen dieses Fehlers, wird wie auch

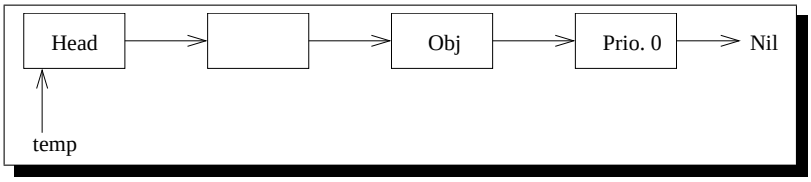

Abbildung 3.16: In folge der Unterbrechung entstandene Situation

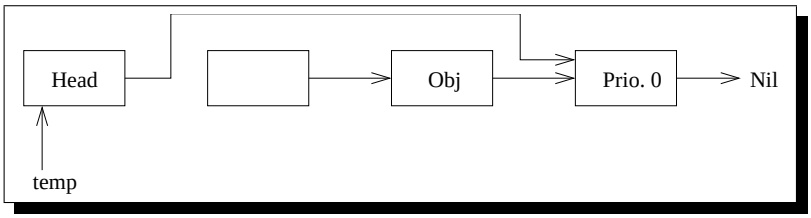

Abbildung 3.17: Endzustand nach Abschluss des Auskettungsvorganges

schon bei der Methode `defer()` durch zu Hilfenahme der Maschineninstruktion *CAS* erbracht. Somit kann ein nebenläufiges Zugreifen auf den Vorgänger, beziehungsweise dessen Veränderung reagiert werden. Der Algorithmus wird geändert und stellt sich hernach folgendermaßen dar.

1. Lesen des Listenkopfes
2. Suchen des Vorgängers des zu entfernenden Elementes
3. *CAS* aufrufen und Element entfernen
4. wenn *CAS* fehlschlug, wiederhole ab 1

Durch diese Arbeitsanweisungen ist sichergestellt, daß ein Element, welches nebenläufig in die Liste integriert wurde, nicht wieder verschwindet. Somit bleibt die Konsistenz der Liste gewahrt und asynchrone Unterbrechungen können den Zustand der Liste nicht negativ beeinflussen.

Ein zweiter Fehler, der auftreten kann, ist um einiges tückischer. Um diesen zu verstehen, muß bekannt sein wie ein Funktionsaufruf funktioniert. Dieser Fehler tritt auf, wenn während des Aufrufes der *CAS*-Funktion eine Unterbrechung geschieht, welche ein Objekt an das zu entnehmende anhängt. Ein kurzer Ausflug zum Thema Funktionsaufruf. Eine Funktion, welche wie die *CAS* Argumente übergeben bekommt, wird wie folgt aufgerufen. Die Argumente der Funktion werden entweder in Register geladen oder auf den Stack (Stapelspeicher) gelegt. Ob die Werte über einen Stack oder über Register an die Funktion übergeben werden, ist prozessor- und sprachenabhänig. Danach erfolgt der Aufruf der Routine. Diese liest die Werte und beginnt mit der Arbeit. Somit können Unterbrechungen zwischen dem laden der Argumente und dem tatsächlichen Funktionsaufruf stattfinden. Das bedeutet, es wird möglicherweise mit veralteten Werten gearbeitet. Das folgende Szenario würde eintreten, wenn

keine Sicherungsmaßnahmen getroffen werden. Die Ausgangsposition ist die im Bild 3.18 dargestellt. Der Zeiger *temp2* verdeutlicht den ins Register oder auf den Stack gelegeten Wert für den Nachfolger.

Eine Unterbrechung erfolgt während des Aufrufes von *CAS*. Ein Element wird unmittelbar an das auszukettende Objekt angehangen. Bild 3.19 verdeutlicht die Situation grafisch.

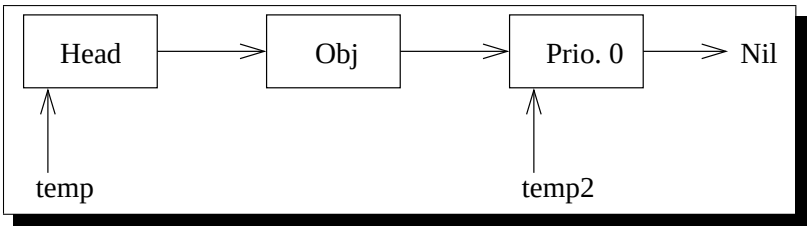

Abbildung 3.18: Ausgangssituation beim Aufruf der Funktion *CAS*

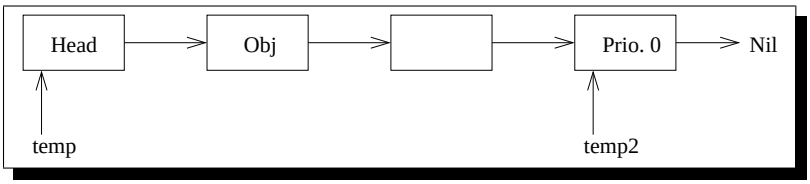

Abbildung 3.19: Von Unterbrechung an auszukettendes Objekt angehangentes Element

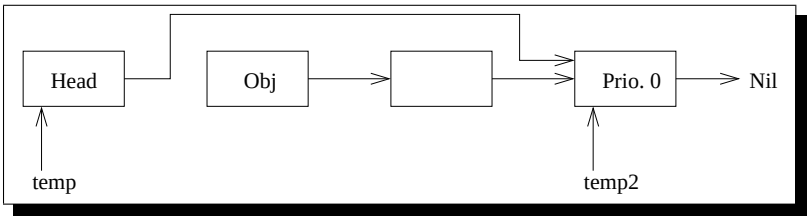

Abbildung 3.20: Versehentliches Ausketten des angehangenen Elementes

Nach dem Zurückkehren aus der Unterbrechung, wird die Ausführung des *CAS* mit dem alten Wert für den Nachfolger fortgesetzt. Als Endzustand entsteht das in Bild 3.20 gezeigt Ergebnis.

Zu sehen ist, daß nicht nur das Objekt, welches entfernt werden soll aus der Liste verschwunden ist, sondern auch das durch die Unterbrechung eingefügte. Es gibt zwei Möglichkeiten, wie dieser Fehler beseitigt werden kann. Zu nennen sind hier

- *DCAS* oder
- merken des Nachfolgers mit einer Nachbehandlung nach dem *CAS*.

Ein *DCAS* funktioniert wie ein *CAS* nur für zwei Adressen. Das *D* in *DCAS* steht für double. Diese Funktion wird aber nur von wenigen Prozessoren unterstützt. Deshalb wird der zweite Lösungsweg angestrebt. Hierbei wird vor dem Aufruf vom *CAS* der Nachfolger des Elementes in einer gesonderten Variablen vermerkt. Nachdem das *CAS* erfolgreich ausgeführt wurde, wird überprüft ob sich der Nachfolger verändert hat. Ist dieser Fall eingetreten, werden die Elemente die fälschlicherweise wieder entfernt wurden, mit Hilfe der Methode `defer()` wieder auf die Liste verbracht. Der Algorithmus ändert sich somit in:

1. Lesen des Listenkopfes
2. Suchen des Vorgängers des zu entfernenden Objektes
3. Merken des Nachfolgers
4. *CAS* aufrufen und Element entfernen
5. wenn *CAS* fehlschlug, wiederhole ab Punkt 1
6. ist Nachfolger nicht mehr der selbe, füge versehentlich entfernte Nachfolger mit `defer()` wieder ein, und gehe dann nach Punkt 6.

Bei dem hier vorgestellten Ablaufplan für das Entfernen von Objekten ist sichergestellt, daß keine Elemente versehentlich von der Liste verschwinden. Wenn die Sequenze vollständig durchlaufen ist, ist die Liste in einem korrekten Zustand. Während der Abarbeitung ergeben sich keine inkonsitenten Zustände. Deshalb kann diese Routine zu jedem Zeitpunkt unterbrochen werden, ohne die Liste damit durcheinander zu bringen.

Der Algorithmus kann noch verbessert werden, damit bei einem Fehlerfall des Punktes fünf nicht wieder vom Anfang der Kette gesucht werden muß. Es wurde die gleiche Technik, wie bei der `defer()`-Funktion angewendet. Somit wird hier auch die „Zeiger auf Zeiger Technik“ angewendet. Dadurch wird gewährleistet, daß bei einem Fehler nur eine geringfügige Neupositionierung des Auskettungspointers vorgenommen werden muß. Der Zeiger muß maximal um die Anzahl Positionen verschoben werden, wie Elemente an den Vorgänger angehangen wurden. Durch diese geringfügige Modifikation der Ablaufsequenz wurde eine Laufzeitverbesserung erreicht, wenn ein Fehler auftritt. Hernach im kommenden Abschnitt werden einige Implementationsdetail angesprochen.

3.2.5.1 Implementationsdetails

Der Quelltext der Methode ist in der Abbildung 3.21 zu sehen. Im Initialisierungsbereich (Zeile 2-4) werden die Zeiger für den Zugriff auf die Liste vorbereitet. Dabei wird einem temporeren Pointer die Adresse des Kopfes der Liste zugewiesen. Wie auch schon bei der Methode `defer()` wird auch hier mit einem Doppelpointer gearbeitet, damit die Funktion möglichst effizient abgearbeitet werden kann. Durch diesen Kniff kann ein Vergleich auf die Manipulation des Listenkopfes unterbleiben und so genau wie eine normale Bearbeitung erfolgen.

```
 1:  void Usher::remove(){
 2:       register Chain **temp=&Head;
 3:       register Chain *entail=(Chain *)0;
 4:       register Chain *item=this;
 5:       do{
 6:            while( *temp != item ){
 7:                 temp = (Chain**)*temp;
 8:            }
 9:       }while( !CAS((Chain*)temp,(Chain*)item,entail=item->select()) );
10:
11:       while( entail != item->select()){
12:            ((Usher*)(item=item->select()))->defer();
13:       }
14:  }
```

Abbildung 3.21: Der Sourcecode der Methode `remove()`

Der *entail*[9] wird später benutzt werden, um den Nachfolger des auszuhängenden Elementes zu sichern. Mit dem *item*-Zeiger wird das Element selber vermerkt, da der *this*-Zeigers des Objektes nicht verändert werden darf und soll. Weiterhin kann erkannt werden, wie der Vorgänger des Elementes gesucht wird (Zeile 6-8). Die umschliessende Schleife (Zeile 5-9) realisiert die Wiederholung falls das *CAS* für das Ausketten fehlschlägt. In dem Aufruf des *CAS* kann außerdem das Sichern des Nachfolgers betrachtet werden. Mit Hilfe des Zeigers *entail* wird dieser zwischengespeichert. Anschliessend wird die Nachbehandlung durchgeführt, sofern diese notwendig wird. Dabei wird im Schleifenkopf (Zeile 11) überprüft ob der Nachfolger verändert wurde. Tritt dieser Fall ein, so beginnt das Einsortieren des Objektes mit der Methode `defer()`. Wenn der Nachfolger wieder mit dem ursprünglichen übereinstimmt, ist die Nachbehandlung abgeschlossen und die Liste wieder in einem korrektem Zustand. Die Konstruktion der Zeile zwölf erledigt mehrere Aufgabe hintereinander. Muß eine Nachbehandlung erfolgen, so wird in der inneren Klammer zuerst eine Position in der Liste weitergegangen, damit das versehentlich entfernte Element genutzt werden kann. Da die Methode `select()` ein *Chain** zurückliefert, muß dieses, da ein Listenelement ein *Usher* ist, gecastet werden, damit der Aufruf von `defer()` funktioniert. Dadurch wird das Objekt wieder in die Liste aufgenommen.

Die diversen Castereien sind notwendig, damit der Compiler die richtigen Funktionaufrufe generiert.

3.2.6 Entfernen des ersten Listenelementes

Die Funktion die hier beschrieben werden soll, erledigt das Entfernen des ersten Listenelementes. Der Name der Routine ist `fetch()`. Dieser leitet sich aus dem englischen ab. Er bedeutet soviel wie holen oder abrufen. Damit wird das Holen

[9] to entail(engl.) = nach sich zeihen

des höchstpriorisierten Objektes von der Kette gemeint.

Nun könnte die Frage aufkommen, warum eine zweite Entnahmefunktion bereitgestellt wird, obwohl die schon beschriebene Methode `remove()` diese Aufgabe ebenfalls erledigen könnte. Wenn beispielsweise die `remove()`-Routine folgendermaßen aufgerufen wird `head()->remove()`, wird das erste Element der Liste entfernt. Da der Algorithmus für das Ausketten von beliebigen Objekten diese erst suchen muß, aber hier immer das erste Objekt entfernt werden soll, bedarf es keiner Suchaktion. Der Algorithmus vereinfacht sich. Die Sequenze die beim Entfernen des am höchsten priorisierten Objektes abgearbeitet wird, sieht folgende Schritte vor.

1. Lesen des Listenkopfes
2. Merken des Nachfolgers des Listenkopfes
3. Ausketten des Listenkopfes mit der Funktion *CAS*
4. Wenn *CAS* fehlschlug, wiederhole bei Punkt 1
5. Ist Nachfolger nicht mehr der selbe, füge versehentlich entfernten Nachfolger mit `defer()` wieder ein, und gehe dann nach Punkt 5
6. Liefere das höchstpriorisierteste Objekt zurück

Hier wurde bereits der korrekte Algorithmus angegeben, wie er sich mit den Sicherungsmaßnahmen darstellt. Der bekannte Fall, daß ein oder mehrere Elemente sich in folge von Unterbrechungen an den Listenkopf anhängen, wird durch die Funktion *CAS* erkannt und abgefangen. Wenn durch Interrupts Elemente an das erste Objekt angehangen werden und diese hernach, durch das Ausketten, sofort wieder von der Liste verschwinden, werden die versehentlich herausgenommenen Elemente, mit Hilfe der schon betrachtete Nachbehandlung, wieder auf der Liste plaziert. Die Fehlersituation wie sie hier auftreten können, sind im Abschnitt 3.2.5 dieses Kapitels beschrieben und sollen deshalb an dieser Stelle nicht wiederholt werden.

Der in der Abbildung 3.22 gezeigte Quelltext veranschaulicht noch einmal den beschriebenen Algorithmus. In der Zeile zwei wird der Pointer *item* angelegt, der im Verlaufe der Abarbeitung das höchstpriorisierteste Element zugewiesen bekommt. Dieser Vorgang findet in der Zeile sechs statt. Außerdem ist dieser Zeiger der Rückgabewert der Funktion. Des weiteren sind in den Zeilen drei und vier zwei weitere Pointer zu erkennen. Hier findet sich auch der bekannte *entail*-Zeiger aus dem vorherigen Abschnitt 3.2.5.1 wieder, der den Nachfolger des Objektes zwischenspeichert und dann als Überprüfungsmöglichkeit in der Nachbehandlung (Zeile 10-12) genutzt wird. Der *temp*-Pointer (Zeile 4) wird benutzt, damit der Inhalt des *item*-Zeigers nicht verloren geht. Er wird einzig und allein in der Nachbehandlung verwendet. Zu den Schleifen Zeile fünf bis sieben und Zeile zehn bis zwölf braucht nicht viel gesagt zu werden. Die *CAS*-Schleife ermöglicht eine Wiederholung, sobald eine nebenläufige Aktion den Kopf der Liste verändert hat. Diese Veränderung kann nur ein Einfügen eines Elemente sein. Dadurch ergibt sich ein neues höchstpriorisiertes Objekt.

```
 1:  Usher* Usher::fetch(){
 2:       register Usher* item=(Usher*)0;
 3:       register Chain *entail=(Chain *)0;
 4:       register Chain *temp;
 5:       do{
 6:            item=(Usher *)head();
 7:       }while ( !CAS((Chain *)&Head,item,entail=item->select()) );
 8:
 9:       temp=item;
10:       while( entail != temp->select()){
11:            ((Usher*)(temp=temp->select()))->defer();
12:       }
13:       return item;
14:  }
```

Abbildung 3.22: Quelltext der Methode `fetch()`

Somit ist dieses zu Entnehmen, welches in einem weiteren Schleifendurchlauf erbracht wird.

Die Nachbehandlungsschleife ist die gleiche wie in der Methode `remove()`. Somit gilt das in Abschnitt 3.2.5.1 beschriebene Verhalten.

Der Abbildung ist weiterhin zu entnehmen das kein Suchvorgang nötig ist. Dadurch wird eine Laufzeitverbesserung gegenüber der Methode `remove()` erreicht. Dies war auch die Motivation für die Erschaffung einer zweiten Entnahmefunktion.

3.2.7 Anwendung des Funktionsumfanges

Die in den vorangegangenen Abschnitten beschriebenen Listenfunktionen müssen mit Bedacht angewendet werden. Zu beachten sind die Unterbrechungsszenarien die in Abschnitt 3.2.3 beschrieben wurden. Aus diesen geht hervor, daß der Aufruf der verschiedenen Methoden kontextabhänig ist. So dürfen nicht einfach die Methoden von der Anwendung aus benutzt werden. Dadurch könnte die Liste in einen inkonsitenten Zustand verfallen. Die Routinen werden einzig und allein über Aufrufe des Schedulers aktiviert. Durch diesen und seine Sicherungsmaßnahmen ist sichergestellt, daß die Funktionen genau das beschriebene Unterbrechungsverhalten aufweisen.

Wie später noch zu sehen ist, wird die `remove()`-Methode nur zum entfernen von aktiven Objekten benutzt. Die Funktion `fetch()` hingegen entfernt beide Objektvariationen, so wie auch die `defer()`-Routine beide Ausprägungen in der Liste anordnet. Warum die Methode `remove()` sich anderst verhält, geht aus dem Kontext des Schedulers hervor.

Aus den gemachten Betrachtungen heraus, ist erkennbar, daß Anwendungen nicht direkt auf die Listenverwaltung zugreifen dürfen.

3.3 Der Scheduler

3.3.1 Entwurf der Schedulerhierarchie

Für das, den Anforderungen gerecht werdende, System ist es erforderlich einen präemptiven prioritätsorientierten Scheduler zu bauen. Das Planungswesen muß hierbei sowohl passive als auch aktive Objekte bedienen können. Dabei soll es aber keine Kenntnis davon besitzen, welcher Objektausprägung ein Objekt angehört. Zur Entscheidungsfindung über die Aktivierung wird einzig die Priorität benötigt. Desweiteren muß das System voll präemptiv arbeiten, wobei präemptiv die unmittelbare Aktivierung eines Objektes bedeutet, wenn es die Kriterien (höchste Priorität) für die Ausführung erfüllt.

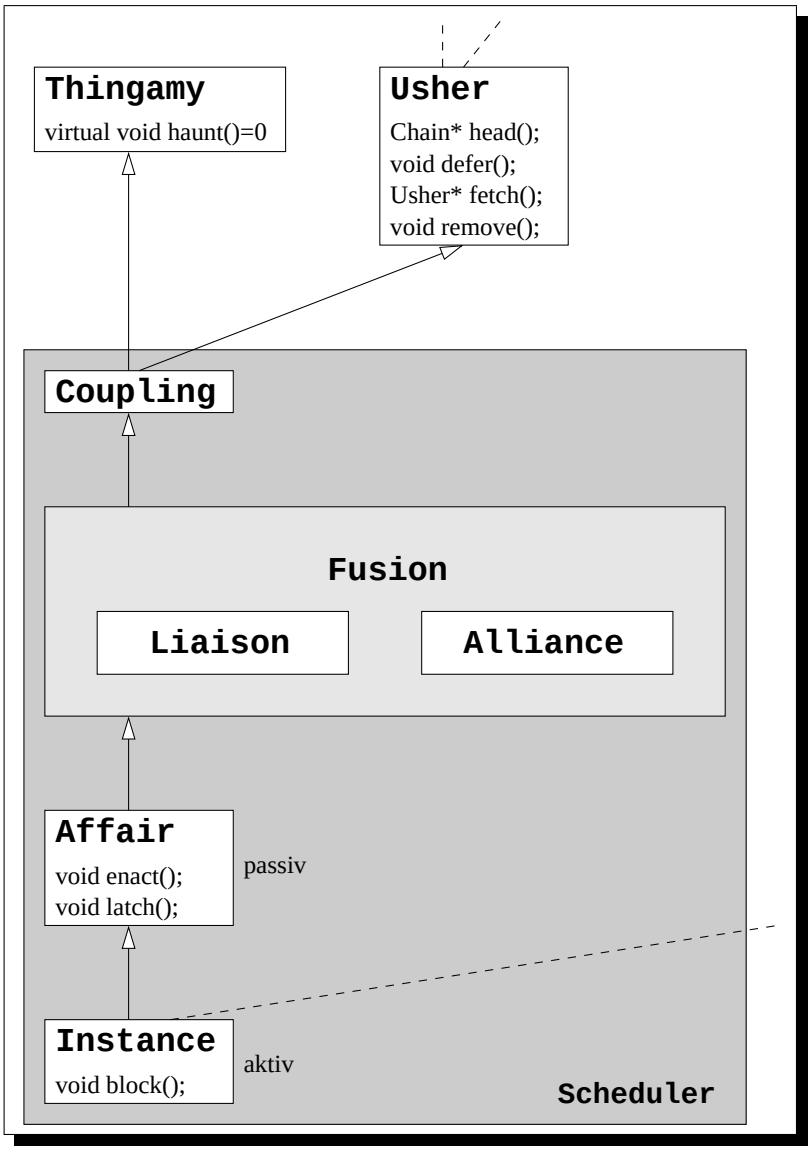

Abbildung 3.23: Vererbungshierarchie des Schedulers

Bei dieser objektorientierten, familienbasierten Implementierung ist der Scheduler ein Teil der Vererbungshierarchie. Wie aus der Abbildung 3.23 zu entnehmen ist, handelt es sich um einen strukturierten, feingranularen Scheduler, der sich aus verschiedenen Klassen zusammensetzt.

Da der Scheduler Objekte aufrufen soll, ohne Wissen über deren Identität, wird ein Mechanismus benötigt, der es ermöglicht, immer die gleiche Funktion anzuspringen, die zu dem richtigen Objekt führt. Für diese Aufgabe wurde die abstrakte Basisklasse *Thingamy*[10] erdacht. Im Bild kann die Deklaration einer Funktion innerhalb der *Thingamy*klasse betrachtet werden. Die pure virtuelle Methode `haunt()`[11] stellt die Aufrufschnittstelle der Objekte dar. Hat der Scheduler also über die Aktivierung eines Objektes entschieden, wird die `haunt()`-Methode ausgeführt. Diese Methode muß durch jedes passive und aktive Objekt definiert werden, damit die Objekte ein spezifiziertes Verhalten aufweisen.

Im Bild ist weiterhin eine zweite Klasse außerhalb des Schedulerrahmens zu sehen. Die bereits bekannte Listenverwaltung, die Klasse *Usher*, findet sich hier wieder. Diese stellt ein verkettungsfähiges priorisietes Objekt zur Verfügung. Daher bildet sie einen Grundbaustein für die zu konstruierenden Klassen.

Die Klasse *Coupling*[12] vereinigt die Listenverwaltung mit einem aufrufbaren Objekt. Es entsteht ein verkettungsfähiges, priorisiertes, aufrufbares Objekt. Ab diesem Zeitpunkt kann bereits von einem passiven Objekt gesprochen werden, da der Informationsgehalt, einer Objektinstanz, diesem Typ entspricht. Das eigentliche passive Objekt liegt jedoch in der Hierarchie weiter oben, da es Schedulingaufgaben übernehmen soll.

Im Bild ist ein Kasten mit den Klassen *Liaison*[13] und *Alliance*[14] zu erkennen. Dieser verdeutlicht die Möglichkeit einer Konfiguration des Planungswesens. Durch eine Einstellung eines *fameFlags*[15] kann eine Konfiguration ausgewählt werden. Dabei kann zwischen verschiedenen Propagationsmodellen gewählt werden. Später werden die Verfahren beschrieben. Der Hauptteil des Schedulings findet hier statt.

Dem Ableitungsgraph folgend, schließt sich die Klasse *Affair*[16] an. Die Klasse definiert die Methoden, die für die Aktivierung von Objekten benötigt werden. Die Methode `latch()`[17] dient dabei dem Starten von Objekten aus dem synchronen Programmablauf heraus. Für das Starten von Objekten aus dem asynchronen Pfad, den Interrupts, ist die Methode `enact()`[18] gedacht. An dieser Stelle ist die Konstruktion von passiven Objekten abgeschlossen. Wenn eine Anwendung passive Objekte verwenden soll, ist die *Affair*-Klasse die Basisklasse all dieser Objekte.

[10] thingamy(engl.) = Dingsda/Irgendwas
[11] to haunt(engl.) = verfolgen, spuken
[12] coupling(engl.) = Kopplung
[13] liaison(engl.) = Verbindung
[14] alliance(engl.) = Bündnis
[15] Konfigurationsschalter
[16] affair(engl.) = Angelegenheit, Affäre
[17] latch(engl.) = Klinke
[18] to enact(engl.) = verordnen, verfügen

Die Klasse *Instance*[19] stellt die Methode `block()`[20] zur Verfügung, welche nur auf den aktiven Objekten aufgerufen werden kann. Sie dient dem Blockieren. Sie ist nötig, um z.B. auf ein Ereignis warten zu können oder einen Prozess zu beenden. Die Kontrolle über den Prozessor wird dadurch abgegeben.

Wie aus dem Ableitungsgraphen ersichtlich, ist die Klasse *Instance* von der Klasse *Affair* abgeleitet, dadurch stehen auch den aktiven Objekten die Aktivierungsmethoden `latch()` und `enact()` zur Verfügung. Die gestrichelten Linien, an den Klassen *Usher* und *Instance*, in der Abbildung zeigen, daß noch weitere Basisklassen für diese Typen existieren. Welche bei der Klasse *Instance* eine Rolle spielen wird in dem Abschnitt 3.3.7 über aktive Objekte erläutert.

Wie der Abbildung weiterhin zu entnehmen ist, sind die passiven und aktiven Objekte Bestandteil des Schedulers. Aufgrund dieser Zusammengehörigkeit erbringen die Klassen *Affair* und *Instance* Schedulingaufgaben.

Zu sehen ist, daß das Planungswesen aus verschiedenen Klassen aufgebaut ist. Hierbei stellen die Klassen *Affair* und *Instance* die äußere Schnittstelle zur Anwendung dar. Das bedeutet, an jenen Stellen kann mit einer weiteren Ableitungfolge zur Spezialisierung der Anwendung begonnen werden. Wie dem Klassendiagramm zu entnehmen ist, stellen die beiden Schnittstellenklassen Methoden bereit. Diese werden genutzt, um in den Scheduler einzutauchen. Da auch jedes weitere abgeleitete Objekt über die Zugangsfunktionen verfügt, brauchen die Objekte nur auf sich selbst die Methoden aufzurufen, um damit in den Scheduler einzutreten.

3.3.2 Schnittstelle des Schedulers

Der Scheduler hat für die verschiedenen Ausführungspfade unterschiedliche Eintrittspunkte. In der Abbildung 3.24 ist zu erkennen, welche Methoden aktiviert werden können, um in den Scheduler einzutauchen. Durch einen Blitz (Interrupt) ist die asynchrone Aktivierung des Planungswesens angedeutet. Der synchrone Programmablauf wird durch eine geschlängelte Linie (Kontrollfaden) symbolisiert.

Objekte werden dem Planungswesen zum Ausführen übergeben. Der Scheduler entscheidet, ob die Abarbeitung sofort oder zu einem späteren Zeitpunkt stattfinden soll. Abhänig vom Schedulingalgorithmus wird diese Entscheidung getroffen.

Für die Aktivierung stehen mehrere Methoden zur Verfügung. In der Skizze ist die Funktion `enact(Exception&)` zu sehen. Durch den Aufruf eines Objektes mit dieser Methode wird der Scheduler betreten, wodurch der Schedulingvorgang angeworfen wird. Das *Exception*-Objekt ist der Parameter, der dem Planungswesen den Grund für die Aktivierung anzeigt. Bei einer Kaskadierung von Unterbrechungen kann anhand des Zustandes des *Exception*-Objektes die mehrfache Aktivierung erkannt und entsprechend reagiert werden.

Die Funktionen `latch()` und `block()` dienen im synchronen Abarbeitungsweg dem Start des Planungswesens. Die Methode `latch()` ermöglicht es, Objekte zu propagieren. Über den Aufruf von `block()` können sich aktive Objekte

[19] instance(engl.) = Vorgang
[20] to block(engl.) = blockieren

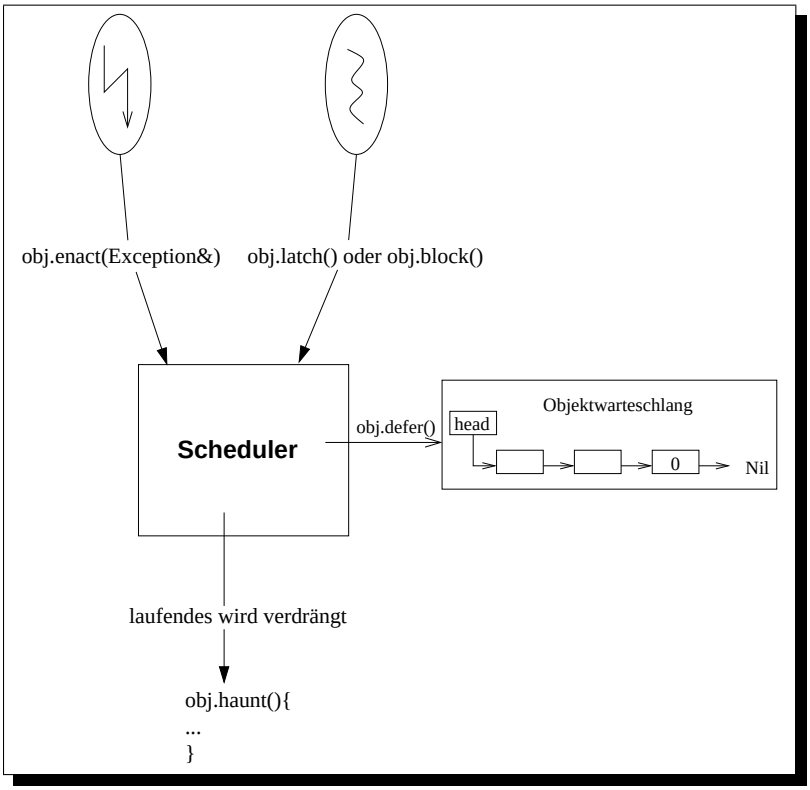

Abbildung 3.24: Eintrittspunkt des Schedulers

blockieren oder beenden. Prinzipell wird eine solche Funktion benötigt, um auf Ereignisse zu warten. Der Prozessor soll dabei aber nicht belegt bleiben, sondern anderen rechenbreiten Objekte für die Ausführung zur Verfügung stehen.

Die abgehenden Pfeile von dem Schedulerkasten zeigen die Ausgänge des zentralen Planungsalgorithmus. Der Ablaufplan des Schedulers entscheidet ob ein Objekt gestartet werden soll, wobei dann das Objekt über die Funktion `haunt()` aktiviert wird. Dabei verdrängt es ein zuvor aktives Objekt vom Prozessor. Soll die Ausführung jedoch aufgeschoben werden und zu einem späteren Zeitpunkt beginnen, muß das Objekt auf die Warteschlage der ausführbaren Objekte verlegt werden. Hierfür wird die Funtion `defer()` benutzt, die das Objekt auf der Liste platziert.

3.3.3 Der Propagationmechanismus

Für die Propagation von Objekten soll hier ein Algorithmus entwickelt werden, der es ermöglicht ein präemptives Scheduling zu verwirklichen. Dabei ist ausschlaggebend, daß keine Prioritätsumkehr auftritt. Des weiteren soll dieser Ablaufplan keine Inkonsistenzen innerhalb der globalen Datenstrukturen hinterlassen.

Zunächst soll jedoch ein einfaches Modell des Algorithmus vorgestellt werden, bei dem auf jegliche Synchronisation von nebenläufigen Aktivitäten ver-

zichtet wird. Es wird davon ausgegangen, daß die gesamte Sequenz unteilbar abläuft, wobei dann keine inkorrekten Datenzustände entstehen können.

3.3.3.1 Ein einfacher Algorithmus

Hier wird ein Algorithmus vorgestellt, der die richtige Reihenfolge der Abarbeitung der Objekte über die Liste regelt. Dabei seien die Objekte priorisiert. Die Bedingung für das Ausführen eines solchen ist, daß seine Priorität grösser ist als die des momentan laufenden Objektes. Wird die Bedingung erfüllt, wird das laufende verdrängt und das andere Objekt gestartet.

```
1:   defer();
2:   if ( head()->prio > life->prio ) {
3:       item = fetch();
4:       item->haunt();
5:   }
```

Abbildung 3.25: Der Schedulingalgorithmus erste Variante

Der Ablaufplan ist in der Abbildung 3.25 im Pseudocode zu sehen. Zunächst wird das Objekt, welches den Scheduler aktivierte, mit Hilfe der Methode `defer()` auf die Liste verschoben. Dadurch ist die richtige Reihenfolge garantiert. Im zweiten Arbeitsschritt wird die Priorität des Listenkopfes (`head()`) mit der des momentan in Ausführung befindlichen Objektes (*life*) verglichen. Ist das Ergebnis des Vergleiches positiv, so muß das momentan ablaufende Objekt verdrängt werden. Dafür entnimmt der Scheduler der Liste das erste Objekt (`fetch()`) und startet es durch den Aufruf der Objektmethode `haunt()`.

Nun ist dieser Algorithmus nicht unbedingt sehr effizient. Wenn beispielsweise ein Objekt eine höhere Priorität aufweist als das momentan laufende, so braucht es nicht den Umweg über die Liste zu nehmen, um propagiert zu werden. Die Schritte des Einfügens und des Entnehmens könnten entfallen. Deshalb wird im nächsten Unterabschnitt eine verbesserte Variante angegeben.

3.3.3.2 Der effizientere Algorithmus

Bei dem in der Skizze 3.26 zu sehenden Ablaufplan wird das Objekt nicht zuerst auf die Liste geschoben. Zu aller erst wird überprüft, ob das Objekt (*this*), welches den Scheduler aktivierte, eine höhere Priorität aufweist als das zur Zeit aktive Objekt (*life*). Hat es eine kleinere Wertigkeit, wobei der Vergleich negativ ausfiel, wird es auf der Liste mit `defer()` platziert. War der Vergleich jedoch positiv, was gleichbedeutend mit der zukünftigen Aktivierung ist, wird die Methode `haunt()` des Objektes aufgerufen.

Prinzipiell ist diese Herangehensweise schneller bezüglich der Propagation von Objekten mit hoher Wertigkeit. Wenn ein Objekt nicht die nötigen Kriterien (höchste Wertigkeit) zur Aktivierung aufweist, muß es auf die Liste. Zu welchem Zeitpunkt dieses erfolgt, ist jedoch in diesem Fall egal.

```
1:   if ( this->prio > life->prio ) {
2:       haunt();
3:   } else {
4:       defer();
5:   }
```

Abbildung 3.26: Der Schedulingalgorithmus effizienter Variante

3.3.3.3 Wechseln des *life*-Zeigers

Ob ein Wechsel des *life*-Pointers erfolgt, hängt von der Objektausprägung ab. Bei einem passiven Objekt bleibt *life* unverändert, weil es auf dem Kontext des aktiven läuft und *life* immer auf den aktiven Kontext verweist. Wurde jedoch die `haunt()`-Funktion von einem aktiven Objekte aufgerufen, wird das momentan aktive Objekt verdrängt. Das Verdrängen erfolgt in zwei Schritten. Als erstes wird das momentan aktive auf die Liste verschoben. Danach wird ein Umschalten der Kontexte vorgenommen. Dafür wird die Methode `resume(`*arg*`)` verwendet. Der Parameter *arg* ist der Kontext, zu dem gewechselt werden soll.

```
1:   void Instance::haunt(){
2:       life->defer();
3:       life->resume(*this);
4:       life = *this;
5:   }
```

Abbildung 3.27: Die `haunt()` Methode eines aktiven Objektes

Nach dem `resume()` setzt das nun arbeitende Objekt an genau der gleichen Stelle fort. Jetzt wird der *life*-Zeiger auf das aktuell laufende Objekt gesetzt. Wenn davon ausgegangen wird, daß diese Arbeitschritte unteilbar abgearbeitet werden, arbeitet diese Methode korrekt.

Im Bild 3.27 ist die Implementierung der Routine `haunt()` für ein aktives Objekt zu sehen. Das Umschalten des *life*-Pointers ist, wie später noch zu sehen, etwas eleganter gelöst, denn es erfolgt implizit bei der Umschaltung der Kontexte. Es sei dazu auf den Abschnitt aktive Objekte 3.3.7.2 verwiesen.

3.3.3.4 Situation beim rekursiven Aufruf des Schedulingvorganges

Wenn aus einem aktiven Objekt heraus der Schedulingvorgang angeworfen wird, entsteht keine kritische Situation, wenn davon ausgegangen wird, daß der Planungsalgorithmus unteilbar abgearbeitet wird.

Problematisch ist nur der Fall falls ein passives Objekt, welches ja auf einem geliehenen Kontext läuft, erneut ein Objekt starten möchte. Der Vorgang des

Scheuling beginnt von Neuem. Da aber nicht die Priorität des passiven Objektes bei dem Ausdruck *life->prio* geliefert wird, sondern die Wertigkeit des aktiven Objektes, welches den Kontext zur Verfügung stellt, spiegelt das Ergebnis des Vergleiches (Abbildung 3.26 Zeile eins) nicht den korrekten Sachverhalt wieder. So muß also eine Möglichkeit gefunden werden die Priorität des passiven Objektes zu bekommen, ohne den *life*-Zeiger zu verändern. Ein Lösung besteht in dem Anheben der Priorität des aktiven Objektes auf die Wertigkeit des passiven Objektes. Dadurch wird bei der Abfrage der Priorität des *life*-Pointers die Priorität des tatsächlich in Ausführung befindlichen Objektes angezeigt.

```
1:   if ( this->prio > life->prio ) {
2:       high = life->prio;
3:       life->prio = this->prio;
4:       haunt();
5:       life->prio = high;
6:   } else {
7:       defer();
8:   }
```

Abbildung 3.28: Der Schedulingalgorithmus erweiterte Variante

Natürlich muß nach der Abarbeitung von Objekten die ursprüngliche Wertigkeit wieder angenommen werden, weshalb sie zwischengespeichert wird. Der Algorithmus ändert sich somit. In der Abbildung 3.28 kann er betrachtet werden. Hier ist das Speichern und das Anheben der Priorität des zugrundeliegenden aktiven Objektes in den Zeile zwei und drei zu sehen. Das Zurücksetzen auf den orginalen Wert erfolgt in der Zeile fünf, so daß das Objekt wieder in seinem ursprünglichen Zustand ist.

3.3.4 Synchronisation im Scheduler

An dieser Stelle wird nicht mehr von einem unteilbaren Ablauf des Schedulingalgorithmus ausgegangen. Vielmehr soll dieser an jeder beliebigen Position unterbrechbar sein, ohne daß ein inkorrektes Verhalten daraus hervorgeht. Der Ablaufplan soll dabei aber keine Inkonsitenzen innerhalb globaler Datenstrukturen hinterlassen.

3.3.4.1 Unterbrechungsszenarien

Zunächst muß untersucht werden, welche möglichen Unterbrechungssituationen auf- und eintreten können. Dabei wird davon ausgegangen, daß der normale Ausführungspfad der Anwendung synchron zum Kern des Systems arbeitet, wobei die zentralen Datenstrukturen (Readyliste, protect-Variable) zu diesem Zeitpunkt in einem konsistenten Zustand sind. Eine Unterbrechung, die ein asynchrones Ereignis darstellt, arbeitet demzufolge asynchron, daher kann in einem solchen Fall nicht von einem gesicherten Zustand des Kerns und der

Systemstrukturen ausgegangen werden. Die hier aufgeführte Aufzählung zeigt, welche Unterbrechungsmöglichkeiten zu behandeln sind.

1. asynchron unterbricht synchron und
2. asynchron unterbricht asynchron

Prinzipell kann an jeder beliebigen Position während der Abarbeitung von Programmcode eine Unterbrechung aufteten. Dies ist auch der Fall, wenn bereits eine Unterbrechung aktiv ist. In einer solchen Situationen wird von der Kaskadierung von Interrupts gesprochen. Hiermit wird der zweite Punkt der Aufzählung gemeint. Der erste Punkt, die Unterbrechung des synchronen Ausführungspfades, spiegelt den eher allgemeinen Fall wieder, da eine Kaskadierung von Unterbrechungen nur in Systemen mit großer Interruptbelastung vorkommt.

In den folgenden Abschnitten wird beschrieben wie die Synchronisation der nebenläufigen Aktivitäten behandelt werden.

3.3.4.2 Der Schutzmechanismus *Locker*

Kritische Abschnitte, wie sie im Abschnitt 3.2.3 beschrieben wurden, gibt es auch im Bereich des Schedulers. Die Behandlung der kritischen Sequenzen soll mit Hilfe einer Art Schloßvariablen erfolgen. Der Unterschied zu der schon beschriebenen Variante aus Abschnitt 3.2.3.2 ist, daß hier nicht aktiv auf das Freiwerden gewartet wird. An dieser Stelle wird nicht mit Spinlocking gearbeitet.

Einen Mechanismus zum Sichern von Bereichen durch Lockvariablen ist bereits Bestandteil der PURE-Klassenbibliothek. Die Klasse *Locker* ermöglicht die Manipulation und das Abfragen einer Sperrvariable. Folgende Methoden werden dafür zur Verfügung gestellt:

`enter()`[21] Mit dieser Methode wird der kritische Bereich als besetzt gekennzeichnet, wobei die Variable um eins erhöht wird.

`retne()`[22] Die Funktion wird aufgerufen, wenn ein kritischer Abschnitt verlassen werden soll. Die Variable wird um eins erniedrigt.

`avail()`[23] Diese Funktion gibt Auskunft über den Zustand der Variablen und damit über den zu schützenden Bereich. Die Variable wird mit null verglichen und wenn dieser Vergleich positiv ausfällt wird ein `true`[24] geliefert. Entsteht hingegen ein negatives Ergebnis beim Vergleich wird `false`[25] zurückgegeben. Über den Rückgabewert der Funktion kann somit angezeigt werden, ob der kritische Abschnitt belegt oder frei ist.

Für den Schutz im Scheduler wird somit eine globale Lock[26]-Variable verwendet. Ihr Name ist *protect*[27]. Das Betreten des kritischen Bereiches erfolgt mit *protect.enter()* und das Verlassen wird durch *protect.retne()* erbracht.

[24] true(engl.) = wahr
[25] false(engl.) = falsch
[26] lock(engl.) = Sperre
[27] to protect(engl.) = sichern

Das Abfragen auf Verfügbarkeit des Abschnittes erfolgt mit dem Aufruf von *protect.avail()*.

3.3.4.3 Fein- vs grobgranulares Sperren

Bei dem Sperren von Bereichen wird zwischen grob und fein unterschieden. Unter den groben Sperrmechanismen werden große Bereiche verstanden. In diesen können eine Menge an Aufgaben erledigt werden. Ein Nachteil ist, daß während dieser Zeit das System keine Unterbrechungen bearbeiten kann. Diese müssen bis zum freiwerden des Abschnittes verzögert werden. Sind diese Abschnitte, wie beim feingranularen Sperren, jedoch eher kurz, kann auf Ereignisse schneller reagiert werden. Ein reaktionsfreudiges System entsteht.

Das System, welches hier vorgestellt werden soll, soll möglichst reaktionsfreudig sein, daher wird die Strategie des feingranularen Sperrens angewendet.

3.3.4.4 Verhalten bei verschlossenem kritischen Abschnitt

Im Scheduler wird vor dem Betreten eines kritischen Abschnittes die Variable gesetzt und beim Verlassen wieder zurückgenommen. Im synchronen Ausführungspfad braucht dabei nicht überprüft zu werden, ob die Variable gesetzt ist, da der Scheduler nur einmal synchron zum Kern betreten werden kann. Wenn das Planungswesen also von der Anwendung aus aufgerufen wird, ist die Konsitenz der Steuer- und Datenstrukruren gewährleistet und der kritische Bereich ist frei.

Wenn hingegen aus dem asynchronen Programmablauf heraus das Planungswesen aktiviert wurde, muß vorher getestet werden ob der kritische Abschnitt verfügbar ist, weil womöglich der Scheduler schon in Betrieb ist. Wenn der Test einen verschlossenen Bereich aufweist, muß die Unterbrechung zurückgestellt werden, was gleichbedeutend mit dem Einsortieren des Interruptobjektes in der Liste ist.

3.3.4.5 Synchronisationsszenarien und kritische Bereiche

Der Schedulingalgorithmus, wie er oben beschrieben wurde, läuft unter den gegebenen Bedingungen, daß Unterbrechungen zugelassen sind, nicht mehr unteilbar ab. Daher können sich in der Abfolge der Befehle Probleme ergeben, wenn Unterbrechungen auftreten. Die sich ergebenden Inkonsitenzen müssen geeignet behandelt werden, damit das System korrekt arbeitet.

3.3.4.5.1 Der erste kritische Abschnitt

Wird zurückschauend noch einmal der Algorithmus aus der Abbildung 3.28 betrachtet, ergibt sich ein kritischer Abschnitt, der die Zeilen eins bis drei umfaßt. Weshalb handelt es sich hier um einen solchen Bereich? Das Problem liegt in der Teilbarkeit des Vergleiches und der darauffolgenden Zuweisung der Priorität des Objektes an *life->prio*. Es kann zu jedem beliebigen Zeitpunkt eine Unterbrechung dazwischenkommen. Versucht diese ihrerseits ein Objekt zu propagieren, kann dieses ungehindert abgearbeitet werden, obwohl

momentan ein Objekt im Scheduler ist. Wenn so eine Situation eintritt, und das Objekt, welches sich im Scheduler befindet, eine höhere Priorität hat, wird von Prioritätsumkehr gesprochen. Diese gilt es jedoch zu vermeiden. In der Abbildung 3.29 kann der Ablaufplan betrachtet werden, der die Unteilbarkeit der Sequenz gewährleistet. Der Schutz des kritischen Abschnittes wurde mit der globalen `protect`-Variablen erreicht.

```
 1:   if ( protect.avail() ) {
 2:       protect.enter();
 3:       if ( this->prio > life->prio ) {
 4:           high = life->prio;
 5:           life->prio = this->prio;
 6:           protect.retne();
 7:           haunt();
 8:           life->prio=high;
 9:       } else {
10:           protect.retne();
11:           defer();
12:       }
13:   } else {
14:       defer();
15:   }
```

Abbildung 3.29: Der Planalgorithmus mit dem Schutz kritischer Abschnitte

Zuerst wird überprüft ob der kritische Bereich frei ist. Bei einem synchronen Eintritt in der Scheduler ist dieser Vergleich immer positiv. Erfolgte jedoch eine Unterbrechung während ein anderes Objekt im kritischen Abschnitt des Schedulers verweilt, wird das Interruptobjekt auf der Liste platziert (Zeile vierzehn), da der Bereich belegt ist. Wenn der Abschnitt bei einer Unterbrechung jedoch frei ist, kann diese ungehindert propagieren.

Wenn der Bereich betreten werden kann, wird dies mit Hilfe der Methode des *Locker* `protect.enter()` getan. Danach erfolgt der Vergleich, ob die Priorität des Objektes ausreichend für die Propagation ist. Ist dies zutreffend, erfolgen die bereits bekannten Arbeitsschritte. Das Verlassen des kritischen Abschnittes erfolgt unmittelbar nach dem Setzen der neuen Priorität. Somit ist die Wertigkeit des zu startenden Objektes gültig. War die Wertigkeit des Objektes nicht ausreichend, muß der kritische Abschnitt dennoch verlassen und das Objekt demzufolge auf die Liste verbracht werden (Zeile zehn- elf).

Durch kritische Abschnitte entstehen natürlich auch weiterhin Probleme. Wie z.B. verhält es sich mit Objekten die eine höhere Wertigkeit aufweisen und auf der Liste landen, weil die momentan im Scheduler befindlichen Objekte den kritischen Bereich halten. Daher muß, nachdem der kritische Abschnitt verlassen wurde, auf die Liste gesehen werden, um eine mögliche Prioritätsumkehr auszuschliessen. Ist das erste Listenelement niedriger priorisiert als das Objekt im Scheduler, kann es ausgeführt werden. Dafür wird das Objekt mit dem Aufruf `haunt()` gestartet. Hat das erste Objekt der Liste jedoch eine ausreichende,

also höhere, Wertigkeit muß dieses abgearbeitet werden. Dafür muß das Objekt, welches im Scheduler ist, auf die Liste und das erste von der selben. Danach muß dieses zur Ausführung gebracht werden.

```
 1:   if ( protect.avail() ) {
 2:       protect.enter();
 3:       if ( this->prio > life->prio ) {
 4:           high = life->prio;
 5:           life->prio = item->prio;
 6:           protect.retne();
 7:           if ( this->prio > head()->prio ) {
 8:               haunt();
 9:           } else {
10:               protect.enter();
11:               defer();
12:               protect.retne();
13:           }
14:           life->prio=high;
15:       } else {
16:           protect.retne();
17:           defer();
18:       }
19:       boost();
20:   } else {
21:       defer();
22:   }
```

Abbildung 3.30: Der Schedulingalgorithmus mit optimistischen Verfahren

Die erweiterte Version des Algorithmus ist im Bild 3.30 zu sehen. Hier wird dem zusätzlichen Überprüfungsvorgang, der durch den kritischen Bereich hervorgerufen wurde, Rechnung getragen. In der Zeile sieben kann der nachträgliche Vergleich betrachtet werden. Wie schon gesagt wurde, muß bei einem negativen Vergleichsergebnis das Objekt auf die Liste und ein anderes ausgeführt werden. Damit ein anderes Objekte ausgeführt werden kann, muß eine Nachbehandlung existieren. In der Zeile neunzehn der Abbildung 3.30 ist die Funktion `boost()` zu sehen, die diese Aufgabe erfüllt. Weiterhin muß immer nach dem Beenden eines Objektes eine Nachbehandlung erfolgen. Warum dies so ist, wird in Abschnitt 3.3.4.6 erschöpfend behandelt.

3.3.4.5.2 Der zweite kritische Bereich

Durch den zusätzlichen Vergleich (Zeile sieben Abb. 3.30) und dem daraus womöglich folgenden Einsortieren eines Objektes in die Liste entsteht ein weiterer kritischer Bereich. Ein Szenario bei dem ein nicht vorhersehbares Systemverhalten entsteht, wenn dieser Bereich nicht geschützt wird, soll hier angegeben werden. In dem Bild 3.30 ist der Bereich jedoch schon als kritisch gekennzeichnet. Mit den Mechanismen des *Locker* wird der Aufruf der Methode

`defer()` in der Zeile elf geschützt.

Wenn der Bereich ohne Schutz verbleibt, beginnt eine „Irrfahrt“, wenn sich folgende Situation ereignet. Ein Objekt `A` soll gestart werden. Es taucht demzufolge in das Planungswesen ein und `A` belegt den kritischen Abschnitt. Ein Unterbrechungsobjekt `B` mit einer höheren Priorität als `A` kann aufgrund des verschlossenen kritischen Abschnittes nur auf der Liste platziert werden und `A` kann, nachdem der Interrupt beendet ist, fortfahren. Bei dem Vergleich (Zeile sieben Abb. 3.30) wird festgestellt das `A` nicht ausgeführt werden darf. Somit beginnt `A` sich auf der Liste zu platzieren. Nachdem `A` den Listenkopf gelesen und in einer lokalen Variable gespeichert hat, wobei der Kopf der Liste dem Objekt `B` entspricht, ereignet sich eine Unterbrechung. Diese propagiert ein Objekt `C` mit einer noch höheren Priorität als `B`. Da kein kritischer Bereich belegt ist, kann es ungehindert verarbeitet werden. Da eine Nachbehandlung immer nach dem Terminieren eines Objektes erfolgen muß (siehe dazu 3.3.4.6), wird nun das Objekt `B` von der Liste genommen und verarbeitet, da es eine höhere Priorität als `A` besitzt. Wird dieses dann beendet, brauchen keine weiteren Objekte mehr verarbeitet zu werden, und es wird zu `A` zurückgekehrt. Dieses arbeitet nun mit einem ungültig gewordenen Element, weil das Objekt `B` nicht mehr Bestandteil der Liste ist. Ein nicht definiertes Verhalten des Systems ist die Folge. Aus den geschilderten Gründen ist hier ein kritischer Bereich, der dementsprechend, wie im Bild zu sehen, geschützt werden muß.

3.3.4.5.3 Umgehen des zweiten kritischen Abschnittes

Zu einem späteren Zeitpunkt in der Entwicklung der Algorithmen zum Planungswesen wurde eine Möglichkeit erwogen den zweiten kritischen Abschnitt zu eleminieren. Hier soll kurz beschrieben werden, wie diese Variation des Ablaufplans arbeitet, und welche Vor- beziehungsweise Nachteile daraus erwachsen.

```
...
6:      protect.retne();
7:      boost();
8:      haunt();
9:      life->prio=high;
...
```

Abbildung 3.31: Der Schedulingalgorithmus ohne zweiten kritischen Abschnitt

Zunächst ist der Punkt, wie kann der zweite kritische Abschnitt vermieden werden? Dabei fällt sofort der Vergleich in Zeile sieben der Abbildung 3.30 auf. Wenn dieser ein negatives Ergebnis liefert, muß das Objekt auf die Liste, wobei das Einsortieren innerhalb eines kritischen Bereiches stattfindet. Danach erfolgt die Nachbehandlung, um weiter Objekte zu propagieren. Somit muß immer, wenn ein Objekt auf der Liste steht, welches eine größer Priorität besitzt, jenes

zur Ausführung gebracht werden. Erreicht werden kann das, indem einfach die Nachbehandlung gestartet wird. Wie zu sehen ist, darf einfach die Nachbehandlung `boost()` aufgerufen werden, bevor ein Objekt verarbeitet wird. Stehen auf der Liste keine höherpriorisierten Objekte, kehrt diese zurück und das Objekt wird mittels der Methode `haunt()` gestartet. Andernfalls werden zuerst die Objekte der Readyliste propagiert. Im Bild 3.31 kann betrachtet werden, wie die Zeilen sieben bis vierzehn der Abbildung 3.30 ersetzt wurden.
Die Vorteile dieser Methode sind klar:

- Ein kritischer Abschnitt weniger
- Sparen eines Einfügevorganges, dadurch schnellere Propagation der Listenobjekte
- Der erzeugte Binärcode ist kleiner (siehe Kapitel 4.1)

Die Nachteile sind jedoch nicht zu unterschätzen:

- Rekursion des Propagationsalgorithmus
- Benötigt größere Stacks

Da die Nachbehandlung im Grunde genommen den selben Propagationsmechnismus benutzt (siehe Abschnitt 3.3.4.6) kann eine Rekursion entstehen. Diese Rekursion ist jedoch nicht endlos. Unterbrechungen sind die einzigen Objektlieferanten, bei denen das Szenario des erneuten Aufrufes des Planungswesens vorkommt. Wird angenommen, daß jeder Interrupt nur ein Objekt propagieren darf, ist die maximale Rekursionstiefe die Anzahl der Unterbrechungen. Ist dieser Wert bekannt, kann berechnet werden, wie groß maximal der Stapelspeicher sein muß, damit es nicht zu einem Überlauf kommt. Kann die Anzahl der Objekte, die aus Unterbrechungen hervorgehen, nicht bestimmt werden, darf die Rekursion nicht auftreten und daher muß dann der zweite kritische Abschnitt bestehen bleiben. Durch die mögliche Rekursion ergibt sich ein erhöhter Stackverbrauch. Natürlich müssen dafür mehr Resourcen zur Verfügung gestellt werden, wodurch diese Variante des Planungswesens sicherlich nur bedingt in Systemen mit wenig Speicher einsetzbar ist.

3.3.4.5.4 Ein dritter kritischer Bereich

Nun zu dem letzten aber nicht minder kritischen Bereich: Das Umschalten zweier Kontexte. In einem der vorangegangenen Abschnitte 3.3.3.3 wurde beschrieben, wie der Wechsel von einem Kontext zu einem andern innerhalb der Methode `haunt()` eines aktiven Objektes durchgeführt wird. Wird dieser Vorgang unteilbar abgearbeitet treten keine Probleme auf. Kommt hingegen eine Unterbrechung dazwischen, die zur Folge hat, daß ein weiteres aktives Objekt gestartet wird, ist das Systemverhalten nicht mehr definiert.

Jetzt wird beschrieben, wann dieser Fehler eintreten kann und wie er beseitigt wird. Folgende Situation liegt vor. Momentan arbeitet ein Objekt A. A startet ein aktives Objekt B mit einer höheren Priorität. Daraufhin entscheidet

das Planungswesen, A muß verdrängt und durch B ersetzt werden. Nun beginnt A sich auf der Liste zu platzieren, damit es zu einem späteren Zeitpunkt wieder aktiviert werden kann. Tritt hier eine Unterbrechung auf, die ein aktives Objekt C mit einer noch höheren Wertigkeit als B starten möchte, kann es ungehindert den Scheduler durchlaufen und C wird propagiert. Da hier wieder ein aktive Objekt nach Abarbeitung verlangt, muß wiederum A verdrängt werden, weil es noch immer den Prozessor hält. Da A möglicherweise bereits auf der Liste steht und ein nochmaliges Einfügen eine Zerstörung der Readyliste bedeutet, darf dieser Vorgang nicht geschehen. Daher wird mit Hilfe des *Locker* ein Schutzring um die Anweisungen des Einfügens und des Kontextwechselns gelegt. In der Abbildung 3.32 kann die so erweiterte Methode `haunt()` der aktiven Objekte betrachtet werden. Das Umsetzen des *life*-Pointers erfolgt hier noch explizit. Im Abschnitt 3.3.7.2 wird beschrieben, wie dies automatisch beim Wechseln der Kontexte vollzogen wird.

```
1:   void Instance::haunt(){
2:       protect.enter();
3:       life->defer();
4:       life->resume(*this);
5:       life = *this;
6:       protect.retne();
7:   }
```

Abbildung 3.32: Die `haunt()` Methode eines aktiven Objektes mit Schutzmechanismus

3.3.4.6 Die Nachbehandlung

3.3.4.6.1 Der Grund für die Nachbehandlung

Es gibt zwei Situationen in denen eine Nachbehandlung erfolgen muß. Durch die Herangehensweise beim verschlossenen kritischen Abschnitt wird sie nötig, sobald ein solcher verlassen wird. Nach dem Beenden von Objekten muß ebenfalls so verfahren werden.

Das erneute Anwerfen des Planungsvorganges muß geschehen, sobald ein Objekt nach dem kritischen Abschnitt nicht mehr das höchstpriorisierteste Objekt im System ist. Der Fall tritt ein wenn ein Objekt `A` im kritischen Abschnitt verweilt. Ein Unterbrechungsobjekt `B` kommt auf die Liste, obwohl `B` eine größere Priorität hat als `A`, weil der kritische Bereich verschlossen ist. Sobald `A` diesen Bereich verläßt muß auf die Liste gesehen werden und gegebenenfalls entsprechend reagiert werden. Folglich muß eine Nachbehandlung angeworfen werden.

Die zweite Situation ist eine recht einfache. Folgendes Szenario macht eine Nachbehandlung unumgänglich. Eine Anwendung `A` startet ein Objekt `B` mit einer hohen Priorität. Das Objekt `B` wird ausgeführt. Während der Abarbeitung werden Unterbrechungsobjekte `C1-Cn` oder Objekte die `B` propagieren möchten,

auf der Liste platziert, sofern diese eine kleinere Wertigkeit haben als `B`. Diese können aber eine höhere Priorität als die Anwendung `A` besitzen. Wenn nun das Objekt `B` beendet ist, müssen zuerst die Objekte `C1-Cn` verarbeitet und erst danach zur Anwendung `A` zurückgekehrt werden. Aus den genannten Gründen muß immer eine Nachbehandlung erfolgen, sobald ein Objekt terminiert. Ob dann tatsächlich Objekte verarbeitet werden, hängt von der Priorität des vorherigen Objektes (hier `A`) ab und dem Zustand der Listenobjekte bzw. deren Prioritäten.

```
 1:  void Liaison::boost(){
 2:      while ( head()->prio > life->prio ) {
 3:          protect.enter();
 4:          if ( head()->prio > life->prio ) {
 5:              item = fetch();
 6:              high = life->prio;
 7:              life->prio = item->prio;
 8:              protect.retne();
 9:              if ( item->prio > head()->prio ) {
10:                  item->haunt();
11:              } else {
12:                  protect.enter();
13:                  item->defer();
14:                  protect.retne();
15:              }
16:              life->prio=high;
17:          } else {
18:              protect.retne();
19:      }
20:  }
```

Abbildung 3.33: Die Nachbehandlungsmethode `boost()`

3.3.4.6.2 Der Algorithmus

Der Algorithmus für die Nachbehandlung kann der Abbildung 3.33 entnommen werden. Ins Auge fällt sofort die große Ähnlichkeit zum Propagationsalgorithmus. Prinzipell muß ja auch das gleiche Verhalten vom Scheduler verlangt werden. Ein geringer Unterschied zum vorangegangenen Ablaufplan stellt hier nur das Arbeiten mit dem Pointer *item* dar, der auf das auszuführende Objekt zeigt. Bei dem Propagationsalgorithmus aus den vorherigen Abschnitten war dies implizit der *this*-Zeiger, weshalb er nicht mit angegeben werden mußte. Diesem *item*-Zeiger wird in der Zeile fünf (Abb. 3.33) das erste Objekt der Liste zugewiesen, wenn über die Ausführung entschieden wurde. Danach erfolgen die gleichen Schritte wie beim Planungsalgorithmus.

Dem Bild ist weiterhin zu entnehmen, daß eine Schleifenkonstruktion (*while*) vorliegt. Dies begründet sich in der Annahme, daß nicht nur ein Objekt während

einer Abarbeitung eines anderen auf der Liste platziert wurde. Durch die Zuhilfenahme dieser Anweisung wird eine Verarbeitung sämtlicher zu Buche stehender Objekte erreicht.

Wie schon gesagt, ähneln sich die beiden Befehlsfolgen der Nachbehandlung und der Planung sehr. Daher werden diese aus Gründen der besseren Lesbarkeit, Wartbarkeit und nicht zuletzt der Effizenz (insbesondere der Verringerung der Codegröße) wegen vereinheitlicht.

```
1:   bool Liaison::ready(){
2:       bool ret;
3:       protect.enter();
4:       if ( !(ret=(*this > life->prio)) )
5:           protect.retne();
6:       return ret;
7:   }
```

Abbildung 3.34: Die Methode `ready()`

```
1:   bool Liaison::patly(){
2:       bool ret;
3:       if ( ret=protect.avail()){
4:           protect.enter();
5:           chip.enable();
6:           if ( !(ret=(*this > life->prio)) )
7:               protect.retne();
8:       }
9:       return ret;
10:  }
```

Abbildung 3.35: Die Methode `patly()`

3.3.4.7 Modularisierung des Planungsalgorithmus

Zunächst wird der kompakte Planungsalgorithmus mit dem Ablaufplan der Nachbehandlung verglichen. Hier kann festgestellt werden, daß das Betreten des kritischen Abschnittes und das Vergleichen mit *life* gleich ist. Daher wird daraus eine Funktion gemacht, die über den Rückgabewert Auskunft gibt, ob der kritische Bereich betreten werden konnte und ob das Objekt eine höhere Priorität aufweist als *life*. Diese Methode gibt es in zwei Ausführungen. Einmal für den Fall, daß Objekte asynchron den Scheduler betreten (Methode `patly()`), also die Abfrage, ob der kritische Bereich belegt ist, erfolgen muß, und zum anderen natürlich ohne die Abfrage, wenn der Scheduler synchron

```
 1:  void Liaison::propagate(){
 2:      high = life->prio;
 3:      life->prio = this->prio;
 4:      protect.retne();
 5:      if ( head()->prio > this->prio){
 6:          protect.enter();
 7:          defer();
 8:          protect.retne();
 9:      } else {
10:          haunt();
11:      }
12:      life()->value(high);
13:  }
```

Abbildung 3.36: Die Methode `propagate()`

aktiviert wird (Methode `ready()`). Den Abbildungen 3.34 und 3.35 kann die Implementierung der beiden Funktion entnommen werden. Im Bild 3.35 der Methode `patly()` kann erkannt werden, zu welchem Zeitpunkt bereits wieder Unterbrechungen zugelassen sind. Dies erfolgt sobald ein Interruptobjekt den kritischen Bereich belegen konnte, wodurch garantiert ist, daß keine anderen Objekte den Scheduler betreten. Mit Hilfe der Funktion `chip.enable()` werden sämtliche Unterbrechungen wieder freigeschaltet. Da dies schon zu einem sehr zeitigen Zeitpunkt geschieht, kann dadurch wieder relativ früh auf Unterbrechungen reagiert werden, wodurch ein reaktionfreudiges System entsteht.

Werden weiterhin beide Algorithmen verglichen, ergibt sich ein identischer Verlauf von Zeile vier bis vierzehn in Abbildung 3.30 und von Zeile sechs bis sechzehn bei der Abbildung 3.33. Weil hier beide Abfolgen gleich sind, wird daraus ebenfalls eine Funktion generiert. Die gemeinsame Weiterverarbeitung erfolgt in der Funktion `propagate()` (Abb. 3.36).

Die Unterschiede der beiden Ablaufpläne (Abb. 3.30 und 3.33) werden in einzelnen Funktionen extra behandelt. So muß z.B. die Funktion `boost()` überarbeitet werden. In dem folgenden Abschnitt wird das Verhalten beziehungsweise der Aufruf der verschiedenen Methoden beim synchronen und asynchronen Programmpfad erläutert.

3.3.4.8 Aktivierung aus dem synchronen Programmpfad

Nun kommt sicherlich die Frage auf, wie spielen diese Funktionen nun genau zusammen? Wird noch einmal die Schnittstelle des Schedulers betrachtet, so muß hier bei dem synchronen Abarbeitungspfad die Methode `latch()` implementiert werden. Wenn also ein Objekt `latch()` aufruft, ergibt sich folgendes Bild. Zunächst ist klar, daß die `ready()`-Funktion gestartet werden muß. Sie überprüft ob der kritische Bereich frei ist und ob die nötige Priorität vorhanden ist. Können beide Fragen mit ja beantwortet werden, so ist der kritische Bereich gesetzt und die Funktion liefert als Ergebnis den Wahrheitswert *true* zurück.

```
1:   void Affair::latch(){
2:       if (!ready() ) {
3:           defer();
4:       } else {
5:           propagate();
6:           boost();
7:       }
8:   }
```

Abbildung 3.37: Die Methode `latch()`

Anhand jenes Wertes erfolgt die weitere Verarbeitung. In dem Fall *true* wird die Funktion `propagate()` gestartet, die das Objekt propagiert. Hat das Objekt terminiert, erfolgt die Nachbehandlung innerhalb der Methode `boost()`. Wenn der Rückgabewert der Funktion `ready()` jedoch *false* ist, bedeutet dies für das Objekt eine Einsortierung in die Liste, damit es zur späteren Verarbeitung bereit steht. Das Objekt hatte folglich keine höhere Priorität als das momentan in Ausführung befindliche Objekt *life*. Die Abbildung 3.37 zeigt die Implementierung der Routine `latch()`. Das entsprechende Verhalten der Funktion kann daran nachvollzogen werden.

3.3.4.9 Asynchrone Aktivierung des Schedulers

Bei der asynchronen Aktivierung des Schedulers wird die Methode `enact()` aufgerufen. Hierdurch taucht das Interruptobjekt in das Planungswesen ein. Die Funktion `enact()` übernimmt dabei die weitere Verarbeitung des Objektes. In der Abbildung 3.38 ist die Routine gezeigt. Dem Bild kann entnommen werden, daß das übergebene *Exception*objekt für die Auswertung des Vergleiches mit herangezogen wird. Liefert der erste Teilausdruck bereits den Wahrheitswert *true*, ist schon ein anderer Interrupt in der Behandlung. Das Ergebnis ist eine umgehende Verschiebung des Objektes auf die Liste und hernach das Verlassen des Planungswesen, damit das unterbrochene Objekt fortfahren kann. War noch keine andere Unterbrechung am Werk (der eher allgemeine Fall) wird die Routine `patly()` betreten.

Ähnlich dem synchronen Starten von Objekten muß zunächst versucht werden den kritischen Bereich zu belegen. Dabei wird jedoch vorher nachgeschaut, ob bereits ein Objekt den kritischen Abschnitt beansprucht. Die Situation kann sich ergeben, wenn die Unterbrechung einen anderen Schedulvorgang unterbrach. Kann das Betreten des kritischen Bereiches erfolgen, werden wieder alle Unterbrechungen zugelassen. Der *Locker* ist jetzt die Bremse für auftretende Interrupts, weil diese beim Betreten von `patly()` sofort zurückgewiesen werden, wodurch sie auf die Liste gelangen. Die weiteren Arbeitsschritte sind bereits bekannt.

Der Rückgabewert der Funktion `patly()` zeigt dann an, wie weiter zu verfahren ist. Ergibt also der Gesamtausdruck in der Zeile zwei der Abbildung 3.38

```
1:   void Affair::enact(Exception& exc){
2:       if ( exc.active() || !patly() ) {
3:           defer();
4:       } else {
5:           propagate();
6:           boost();
7:       }
8:   }
```

Abbildung 3.38: Die Methode `enact()`

den Wert *true*, hat das Objekt nicht die Berechtigung ausgeführt zu werden. Ist hingegen der Wahrheitswert insgesamt *false*, erfolgt die Propagation des Objektes mit anschliessender Nachbehandlung. Somit ist die Behandlung von Unterbrechungen durch den Scheduler abgeschlossen.

3.3.4.10 Die optimierte Variante der Nachbehandlung

Die Nachbehandlung kann natürlich von der Vereinheitlichung der Algorithmen profitieren. Ein erster Punkt ist die Anwendung der gleichen Funktionen, wodurch somit ein gleiches Verhalten bei der Propagation erreicht wird. Ein weiteren Fakt ist die bessere Wartbarkeit, da eine Änderung einer Funktion sich immer auf den zentralen Algorithmus sowie auf die Nachbehandlung auswirkt.

Nun aber zu der eher technischen Sicht. Im Bild 3.39 kann die optimierte und zugleich um einiges verständlichere Variante der Methode `boost()` betrachtet werden.

```
1:   void Liaison::boost(){
2:       while ( head()->ready() ){
3:           item=fetch();
4:           item->propagate();
5:       }
6:   }
```

Abbildung 3.39: Die optimierte Methode `boost()`

Die bereits bekannte Methode `ready()` wird mit Hilfe des ersten Listenelementes aufgerufen. Ist die Priorität dieses Objektes größer als die von *life*, hält das Objekt den kritischen Abschnitt. Danach kann es von der Liste genommen werden, ohne die Gefahr, daß ein anderes Objekt momentan den Scheduler erfolgreich durchlaufen kann. Weiterhin wird hierdurch sichergestellt, daß der Zugriff auf die Liste mittels der Funktion `fetch()` nur einmal also nicht nebenläufig stattfindet kann, da das Objekt im kritischen Abschnitt verweilt und

die `fetch()`-Routine nur hier ihre Verwendung findet.

Nachdem das Objekt erfolgreich von der Readyliste entfernt werden konnte, wird es durch den Aufruf *item->propagate()* zur Ausführung gebracht. Hat das Objekt seine Arbeit beendet, kehrt es hierher zurück und weitere Objekt können verarbeitet werden. Durch die Schleifenkonstruktion werden solange Objekte verarbeitet, bis die Priorität der Listenobjekte nicht mehr größer als die *life*-Priorität ist. Wenn dies der Fall ist, wird das Planungswesen verlassen und *life* kann seine ursprüngliche Arbeit fortsetzen.

3.3.5 Konfiguration des Planungswesens

Das Planungswesen kann konfiguriert werden, dieses wurde in der Beschreibung zum Entwurf bereits angedeutet. Die Konfigurationsmöglichkeit betrifft die im Kern residierende Propagationsfunktion `propagate()`. Sie liegt in zwei Versionen vor, wobei aber immer nur eine im System vorhanden sein kann. Somit muß eine Auswahl getroffen werden. Über die Einstellung eines *fameFlags* wird die bevorzugte Propagationsvariante ausgesucht.

Die dem System zur Verfügung stehenden Variationen des Planungsalgorithmus sind folgende:

1. Vorhandensein des zweiten kritischen Abschnittes
2. Wenn nötig – propagiere rekursiv

Mit Hilfe des *fameFlags famePropagate_If_Necessary_Recursiv* kann dann eine der beiden Versionen ausgewählt werden.

Ist das Flag nicht gesetzt, ist die Version mit dem zweiten kritischen Abschnitt für das System bestimmt worden. Dadurch ist die Klasse *Liaison* Bestandteil des Ableitungsgraphes und somit Basisklasse für die *Affair*. Anderenfalls ist *Alliance* die Basis für *Affair*. In diesem Fall wird die Nachbehandlung vor dem Aktivieren eines Objektes angesprungen, wobei dann eine Rekursion entstehen kann. Diese ist jedoch endlich (siehe 3.3.4.5.3).

3.3.6 Passive Objekte

Ein passives Objekt stellt keinen eigenen Kontrollfluß dar. Es ist vielmehr nur eine Prozedur (Stückchen Code), welche zum Ablaufen zu können einen Kontrollfluß benötigt. Somit läuft das passive Objekt auf dem geliehenen Kontext eines aktiven. Es okkupiert für die Zeit seiner Ausführung die Resourcen eines aktiven Objektes. Die Charakteristik entspricht einer Unterbrechung, die auch auf diese Weise vorgeht. Passive Objekte sind unter anderem dazu angedacht, Unterbrechungen vielmehr deren Nachbehandlung[28] zu priorisieren und dadurch eine nach der Priorität geordnete Abarbeitung, durch den Scheduler zu ermöglichen. Durch diese Priorisierung wird erreicht, daß wichtigere Unterbrechungen auch bevorzugt abgearbeitet werden können. Bisher wurde in PURE nur eine Ausführung nach der Reihenfolge des Auftretens der Interrupts unterstützt.

[28] Nachbehandlung = entspricht den aus PURE bekannten Epilogen

Die Basisklasse für ein passives Objekt ist die Klasse *Affair*. Hier ist die Schnittstelle zum Scheduler implementiert, über die die Objekte in das Planungswesen eintauchen, damit sie zur Ausführung gebracht werden können. Dafür stehen zwei verschiedene Funktionen bereit. Zum einen kann ein passives Objekt von einer Anwendung heraus mit Hilfe von `latch()` gestartet werden. Zum anderen ist die Methode `enact()` zum Aktivieren des Objektes aus einer Unterbrechnug heraus bereitgestellt. Daraus folgt, wenn ein passives Objekt benutzt werden soll, muß dieses von der Klasse *Affair* abgeleitet werden. Dabei erbt es sämtliche Methoden, wodurch dem Objekt dann alle Aktivierungsmethoden zur Verfügung stehen.

Der Speicherbedarf eines passiven Objektes begrenzt sich auf wenige Bytes in Bezug auf die Verwaltungsinformationen. In der Abbildung ist ein Speicherauszug eines solchen Objektes zu sehen. Hier kann erkannt werden, daß nur sehr wenige Daten wirklich für die Verarbeitung durch den Scheduler benötigt werden. Der obligatorische Pointer *void *vtp*, in dem Fall das virtuelle Funktionen in der Klasse vorhanden sind, ist ein Zeiger auf eine Tabelle mit Adressen auf Funktionen. Hierüber werden virtuelle Funktionsaufrufe aufgelöst. Der zweite Zeiger im Bild *Chain *link* dient der Verkettung in der Readyliste, sobald ein Objekt auf dieser platziert wird. Der dritte Wert, die Wertigkeit, wird genutzt, um Vergleiche anzustellen, die darüber Auskunft geben, ob das Objekt propagiert oder auf die Liste verbracht wird. Danach sind die spezifischen Daten jedes einzelnen passiven Objektes gespeichert. Diese können von Objekt zu Objekt unterschiedlich sein.

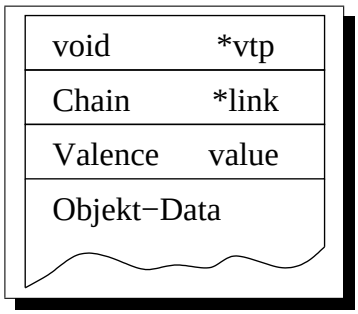

Abbildung 3.40: Das Speicherabbild eines passiven Objektes

Das spezifische Objektverhalten muß in der Methode `haunt()` implementiert werden. Diese Funktion wird durch das Planungswesen aufgerufen, sobald über die Abarbeitung des jeweiligen Objektes entschieden wurde.

3.3.7 Aktive Objekte

3.3.7.1 Entwurf

Bei einem aktivem Objekt handelt es sich um einen autonomen Kontrollfluß, welcher durch den Prozessor direkt ausgeführt werden kann. Dazu ist es nötig, daß ein eigener Prozeßkontrollbock(PCB) zur Verfügung steht. Das Halten des

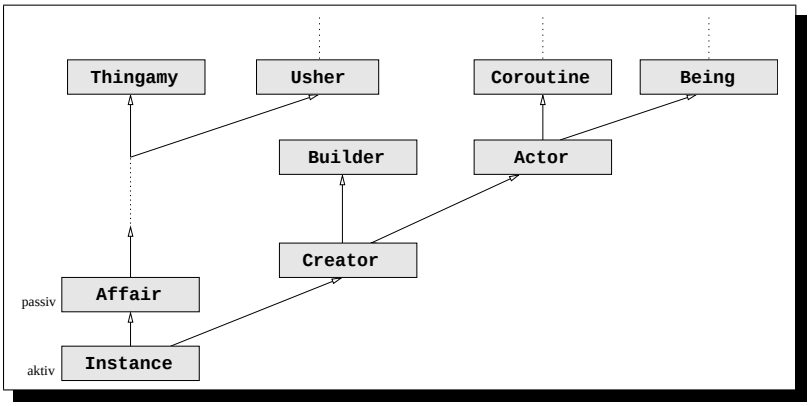

Abbildung 3.41: Vererbungshierarchie eines passiven und aktiven Objektes

PCB und das Multiplexen des Prozessors erbringt die Klasse *Actor*[29] und deren Basisklassen. Somit ist die Klasse *Actor* eine Basisklasse für ein aktives Objekt, welches durch die Klasse *Instance* repräsentiert wird. In der Abbildung 3.41 ist ein Klassendiagramm zu sehen, in dem die Ableitungsfolge der Klasse *Instance* dargestellt ist. *Instance* erbt unter anderem auch von der Klasse *Affair*, wodurch ihr die Aktivierungsroutinen des Schedulers zugänglich sind. Natürlich hat sie damit auch einen Verkettungszeiger und eine Wertigkeit geerbt, welche von den Basisklassen unterhalb von *Affair* bereitgestellt werden.

Ein Kontrollfluß braucht desweiteren einen eigenen Stack. Dieser Stack wird von der Klasse *Builder*[30] erzeugt. Hierbei handlet es sich nicht um einen gewöhnlichen Stack, sondern um einen der an einer alignten[31] Adresse beginnt. Weshalb hier ein spezieller Stack benötigt wird, und welche Vorteile es bringt, wird im Kapitel 3.3.7.2 beschrieben. Für den Entwurf soll nur klar gestellt sein, daß die Klasse *Builder* einen Stack bereit stellt.

Im Bild kann auch die Klasse *Creator*[32] betrachtet werden. Sie sichert die richtige Abarbeitungsreihenfolge der Konstruktoren der Basisklassen *Builder* und *Actor*. Dieses ist wichtig, da der Konstruktor der Klasse *Builder* den Stack erzeugt, welcher im Konstruktor von *Actor* weiter benutzt wird. Im *Actor* wird dann der Stack so vorbereitet, daß ein Objekt später starten kann.

3.3.7.2 Stackbesonderheit

Es wurde bereits erwähnt, daß der Stack eine besondere Rolle spielt. Der Stapelspeicher liegt an einer alignten Adresse. Der Stack erfüllt diese spezielle Voraussetzung, damit aus dieser Adresse herausgerechnet werden kann, welcher Prozess oder welches aktive Objekt momentan auf einem Prozessor abläuft. Dafür wird der *this*-Pointer des Objektes an den Anfang seines Stacks gelegt. Wenn nun der Prozessor arbeitet, kann durch ausmaskieren der Stackadresse

[29] actor(engl.) = Akteur, Schauspieler
[30] builder(engl.) = Baumeister
[31] alignment(engl.) = Anordnung, Einstellung
[32] creator(engl.) = Erschaffer, Schöpfer

der Zeiger auf den Prozesskontrollblock errechnet werden. Dadurch wird eine zusätzliche Pointervariable, die den momentan laufenden Prozess kennzeichnet, überflüssig. Desweitern kann der kritische Bereich des Eintragens des neuen Prozesses und zu diesem Wechseln aufgelöst werden.

In der Abbildung 3.42 ist der Stack und der Prozeßkotrollblock zu sehen. Die Pfeile deuten an, auf welche Adressen die Zeiger verweisen. Zu sehen ist, wie an der obersten Stelle des Stapelspeichers der *this*-Pointer wieder auf den Prozeßkontrollblock zeigt.

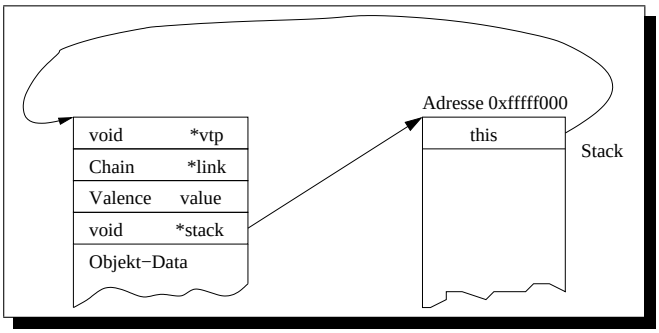

Abbildung 3.42: Stack und Speicheraufbau eines aktiven Objektes

Die Interna des Stacks, die mit der Speicherverwaltung in Verbindung stehen, werden durch die Arbeit [13] von Helge Sichting behandelt. Insbesondere die Realisierung der alignten Adressen, die für den Stack benötigt werden, werden dort erklärt. Bei Fragen auf diesem Gebiet sei auf diese Arbeit verwiesen.

Bei einem Kontextwechsel erfolgt natürlich auch das Umsetzen des Stackpointers. Hiermit wird gewährleistet, daß auch implizit der *this*-Zeiger verändert wird. Dadurch kann ein zusätzlicher Zeiger für das Speichern des momentan laufenden Objektes eingespart werden, und weiterhin das explizite Setzen des Pointers. Eine Einsparung an Speicherplatz und ein schnelleres Wechseln zwischen aktiven Objekten wird dabei erreicht. Ein Nachteil ist jedoch, daß eine spezielle Speicheranforderungsmethode vorhanden sein muß.

3.3.7.3 Das Starten von aktiven Objekten

Das Starten von aktiven Objekten ist eng mit dem Stack verknüpft. Die Initialisierung des Stapelspeichers mit den *this*-Zeiger ist nur der Anfang einer Reihe von weiteren Vorbelegungen. Die Klasse `Actor` und deren Basisklassen nehmen weitere Zuweisungen vor, die im späteren Verlauf ein Starten des Objektes ermöglichen. Hier soll nur kurz beschrieben werden, wie der Stack nach diesen Vorbereitungen aussieht, und wie der Startvorgang vor sich geht.

Auf dem Stack werden nacheinander die Adresse des Objektes, ein beliebiger Wert und die Adresse einer Startfunktion gepackt. Wenn nun ein Wechsel der Kontexte durchgeführt wird, wobei auch die Stacks ausgetauscht werden, erfolgt ein Rücksprung aus dem `resume()` an die Adresse, die zuletzt auf dem

Stapelspeicher liegt. Dies ist genau die Adresse der Startfunktion, welche hiermit angsprungen wird. Die Funktion benötigt nun einen Parameter, der vom Stack geholt wird. Dieser ist ein aktives Objekt, wobei ja die Adresse eines solchen Objektes sich an genau der Position befindet, an der die Funktion ihren Parameter erwartet. Die Funktion ruft dann immer die gleiche Schnittstellenfunktion des Objektes auf, wodurch es startet. Der beliebige Wert auf dem Stack ist eine Rücksprungadresse, falls aus der Startfunktion zurückgekehrt werden soll. Dies muß jedoch verhindert werden, weil es keinen direkten Aufrufer gibt, zu dem es einen Rückweg gibt. Falls doch versucht wird zurückzuspringen, ist das Systemverhalten undefiniert. Der beliebige Wert auf dem Stack, zwischen den Adressen des Objektes und der Startfunktion, ist einzig und allein aus Gründen der Aufrufeigenschaften von Funktionen nötig. Prinzipell könnte hier eine Adresse einer Funktion eingetragen werden, die die Beseitigung eines Objektes vornimmt, welches sich endgültig beenden möchte. Da aber so eine Funktion in PURE nicht existiert, kann es ein beliebiger Wert sein.

```
1:   void kickoff (Being* body) {
2:       for (;;) {
3:           (*body)();
4:       }
5:   }
```

Abbildung 3.43: Der Start eines aktiven Objektes

In der Abbildung 3.43 kann die kickoff[33]-Funktion betrachtet werden. Der Operator () ist die definierte Aufrufschnittstelle eines aktiven Objektes. In dem folgenden Bild 3.44 ist die Implementierung diese Operators zu sehen. Zunächst können mittels der Methode `appear()`[34] einige Vorbereitungen gemacht werden. In dem hier vorgestellten System hat diese Funktion eine besondere Aufgabe. In dem Abschnitt 3.3.7.5 wird darauf näher eingegangen. Die Methode `action()`[35] entspricht der Methode `haunt()` der passiven Objekte, da in dieser das spezifische Objektverhalten programmiert werden muß. Die letze Routine `burial()`[36] wird in diesem System nicht benutzt. Da die Methoden der Klasse *Actor* und deren Basisklassen nicht im Rahmen dieser Arbeit implementiert wurden, sondern nur verwendet, ist dort keine Änderung vorgenommen worden. Möglicherweise stellt sich bei einer Weiterentwicklung auch heraus, daß die Methode `burial()` sehr nützlich ist.

[33] kickoff(engl.) = Start
[34] appear(engl.) = erscheinen, auftauchen
[35] action(engl.) = Aktion, Handlung
[36] burial(engl.) = Begräbnis, Beseitigung

```
1:   void Being::operator () () {
2:       appear(); // initialisiere Prozesszustand
3:       action(); // spring zu dem Eintrittspunkt
4:       burial(); // friere Prozessstatus ein
5:   }
```

Abbildung 3.44: Die Geburtsmethode eines aktiven Objektes

3.3.7.4 Blockieren eines aktiven Objektes

Eine weitere Funktion, die von einem Prozess oder auch einem aktiven Objekt benötigt wird, ist die Blockierungsroutine. Eine solche wird beispielsweise gebraucht, damit beim Warten auf Ereignisse nicht der Prozessor belegt bleibt. Dadurch kann anderen Objekten die Möglichkeit einer Abarbeitung gegeben werden. Das aktive Objekt gibt freiwillig die Resource Prozessor ab.

Nun zählt die Aufgabe des Blockierens zu dem Bereich des Schedulers, weshalb er auch eine solche Funktion anbieten muß. Da die aktiven Objekte Bestandteil des feinstrukturierten Planungswesens sind, ist die Methode `block()` eben genau hier in der Klasse *Instance* angesiedelt und implementiert.

3.3.7.4.1 Schematische Darstellung

Im Bild 3.45 ist der Blockiervorgang eines aktiven Objektes schematisch dargestellt. Zu sehen sind die Readyliste, der Scheduler, als große Blackbox, und natürlich die Objekte. Grob betrachtet spielt sich dabei folgender Ablauf ab. Das Objekt *A* ruft `block()` auf, damit es blockiert. Danach wird ein Objekt *B* von der Readyliste entfernt und zur Ausführung gebracht. Mittels `resume()` wird zu dem Objekt *B* gesprungen, und schlußendlich landet das Objekt in der `action()`-Routine, in der das spezifische Objektverhalten von *B* implementiert ist.

Wird der Vorgang etwas genauer unter die Lupe genommen, muß beachtet werden, daß nur zu einem anderen aktiven Objekt mittels `resume()` gewechselt werden kann. Daher darf aus der Liste nicht irgend ein Objekt ausgewählt werden, sondern es muß das nächste aktive Objekt gesucht und gefunden werden.

Zunächst wird davon ausgegangen, daß immer ein aktives Objekt auf der Liste steht, wodurch folglich die Suchfunktion stets ein Objekt liefern kann. Ist es gefunden, muß es entfernt werden. Die Funktion `remove()` der Listenverarbeitung erbringt das Ausketten des Objektes. Danach wird sofort mit Hilfe von `resume()` zu dem Objekt gewechselt. Nun darf jedoch nicht zur `action()`-Methode umgeschaltet werden, da möglicherweise dieses aktive Objekt, sei sein Name wiederum *B*, nicht die höchste Wertigkeit im System besitzt. Da das Objekt *B* auf der Liste gesucht wurde, impliziert dieses, daß ein oder mehrere Objekte *C0..Cn* mit größerer Priorität als *B* noch vor diesem angeordnet

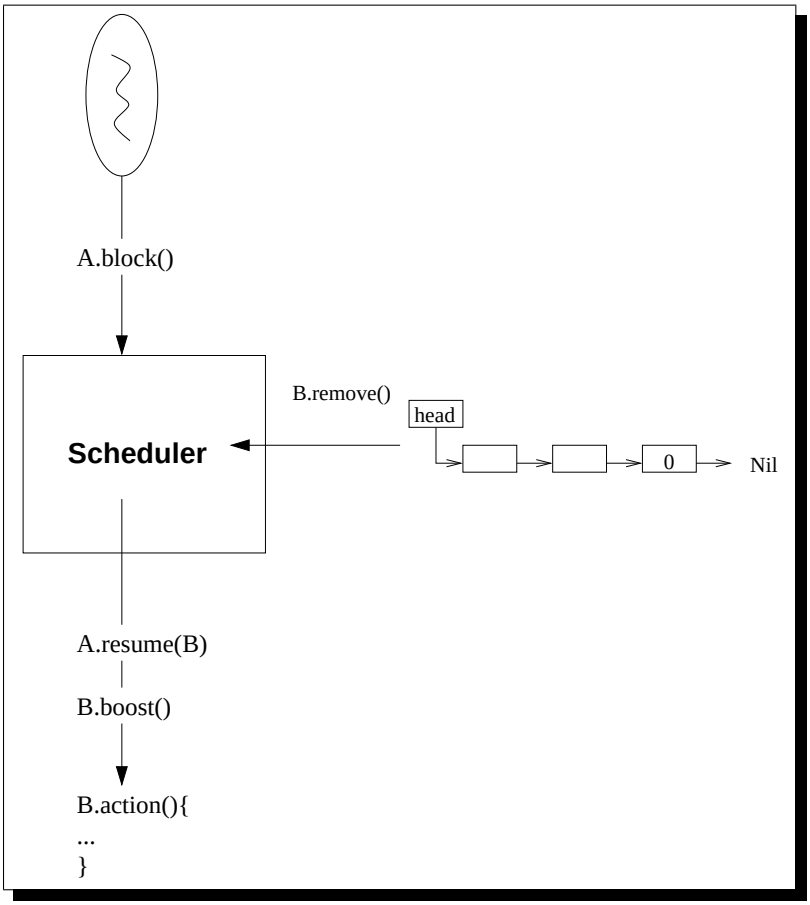

Abbildung 3.45: Schematische Darstellung des Blockierungsvorganges

waren. Somit müssen zuerst die Objekte *C0..Cn* verarbeitet werden. Deshalb wird, bevor die Funktion `action()` des Objektes *B* betreten wird, die schon bekannte Methode `boost()` des Planugswesens aufgerufen, die noch anstehende Objekte propagiert. Bei den Objekten *C0..Cn* kann es sich im Allgemeinen nur um passive handeln, weil durch die Suchfunktion ja das nächste aktive Objekt in der Liste geliefert wurde. Eine Ausnahme kann da nur eine Unterbrechung sein, die ihrerseits ein aktives Objekt auf die Liste gepackt hat. Dies ist aber nicht problematisch, da einfach zu dem Objekt gewechselt wird, wenn es an der Reihe ist.

Nun zu dem Fall daß nicht garantiert werden kann, daß immer ein aktives Objekt auf der Liste verfügbar ist. Entweder dürfen dann die Objekte allgemein nicht blockieren oder es muß dafür gesorgt sein, daß immer ein aktives Objekt bereit ist, welches auf der Liste steht. Ist nicht sichergestellt, daß Objekte nicht blockieren, muß ein Idleprozess[37] verfügbar sein, der dann an die Reihe kommt, wenn alle anderen aktiven Objekte blockiert sind. Dieser hat natürlich eine sehr kleine Wertigkeit, wodurch er nur im Notfall zum Rechnen kommt. Seine

[37] idle(engl.) = Leerlauf

Rechenzeit verbringt er mit dem Propagieren von Objekten, die noch auf der Readyliste stehen. Sollten keine Objekte mehr zu propagieren sein, wird der Idleprozess eine Endlosschleife betreten. Da das System voll präemptiv arbeitet, wird eine Unterbrechung stets durchkommen und verarbeitet werden. Daher ist der Eintritt in die Endlosschleife kein Problem.

3.3.7.4.2 Die technische Sicht auf das Blockieren

Technisch gesehen, muß natürlich die Suche nach einem aktiven Objekt auf der Liste unterstützt werden. Prinzipell kann diese Funktion über zwei Wege erbracht werden.

1. Jedes Objekt speichert seine Objektausprägung in einer Variablen, die über eine Funktion abgefragt werden kann.

2. Ein Objekt liefert sich selbst, wenn es ein aktives Objekt ist. Sofern es ein passives Objekt ist, wird einfach die Suchfunktion des Nachfolgers aus der Liste aufgerufen. Hierbei handelt es sich um eine Rekursion, die spätestens beim Auffinden des Idleobjektes endet.

Beide Varianten führen zum Ziel, und es ist jeweils eine virtuelle Funktion in den Objekten dafür anzubieten. In dem ersten Fall bedarf es jedoch einer Interpretation des Rückgabewertes. Folglich muß immer ein *if*-Statement[38] bei der Suche und der Auswertung des Wertes benutzt werden. Es verlangt also eine Fallunterscheidung, die bei dem Ergebnis „passives Objekt“ weitersucht. Bei der zweiten Herangehensweise wird einfach ein Zeiger auf ein aktives Objekt geliefert. Eine zusätzliche Deutung ist nicht nötig. Der Suchvorgang ist hier implizit durch die Funktion gewährleistet. Wenn ein Zeiger als Ergebnis der Suche feststeht, kann sofort mit dem Ausketten des dazugehörenden Objektes fortgefahren werden.

```
1:   Affair* Instance::get_context(){
2:       return this;
3:   }
1:   Affair* Affair::get_context(){
2:       return ((Affair*)select())->get_context();
3:   }
```

Abbildung 3.46: Die `get_context()` Methoden der beide Objektenvarianten

Für das zugrundeliegende System wurde der zweite Ansatz verwirklicht. Dadurch ergibt sich eine kleine Änderung innerhalb der Klasse *Affair* (passiv), weil diese nun eine virtuelle Funktion benötigt, die die Suchfunktion des

[38] statement(engl.) = Anweisung

Nachfolgers in der Liste aufruft. Der Name der Suchroutine ist `get_context()`. In der Klasse *Instance* (aktiv) wird als Rückgabewert dieser Funktion einfach der *this*-Zeiger geliefert. Somit ist das Ergebnis der Suche ein Zeiger auf ein aktives Objekt. Im Bild 3.46 können die Implementationen der Funktionen, jeweils für aktive und passive Objekte, betrachtet werden. Bei den passiven Objekten wird mit Hilfe der Methode `select()` der Nachfolger bestimmt. Der gelieferte Zeiger muß jedoch erst in einen *Affair*-Zeiger umgewandelt werden, bevor die Suchfunktion aufgerufen werden kann.

Ist ein Objekt gefunden, kann im folgenden Arbeitsschritt das Objekt von der Liste entfernt werden. Danach wird ein kritischer Bereich betreten und ein `resume()` zu dem anderen aktiven Objekt durchgeführt. Der kritische Abschnitt muß benutzt werden, weil das andere Objekt bei dem Austritt aus `resume()` einen solchen verläßt. Es wurde bereits angedeutet, daß die Methoden `enter()` und `retne()` des *Locker* immer als geschlossenes Paar angewendet werden müssen, damit keine unerwarteten und unvorhersehbaren Effekte auftreten. Insbesondere das zu weite Herunterzählen darf nicht vorkommen, weil die Schloßvariable keinen negativen Wert erhalten darf. Deshalb muß, wenn an einer Stelle im System ein solcher Aufrufkomplex um `resume()` existiert, bei jedem weiteren Auftauchen von `resume()` ebenfalls ein Schutzring aus `enter()` und `retne()` gezogen werden. Aus diesem Grund ist auch nach dem `resume()` das Verlassen des kritischen Abschnittes Bestandteil dieser Routine. In der Abbildung 3.47 ist die Realisierung der gesamten `block()`-Funktion zu sehen.

```
1:   void Instance::block(){
2:       register Instance* item=head()->get_context();
3:       item->remove();
4:       protect.enter();
5:       resume(*item);
6:       protect.retne();
7:   }
```

Abbildung 3.47: Die `block()` Methode eines aktiven Objektes

3.3.7.5 Besonderheit bei der Aktivierung

Eine Besonderheit in der Aktivierung von aktiven Objekten liegt in der Tatsache begründet, daß sie, von einem sich blockierenden Objekt aus, gestartet werden können. Da in so einem Fall nicht das Objekt als solches die höchste Priorität hat, darf es nicht ohne Umschweife in seine `action()`-Routine springen, welche das objektspezifische Verhalten verwirklicht.

Handelt es sich bei dem zu aktivierenden Objekt, um eines welches schon einmal gelaufen ist, also nur unterbrochen oder blockiert war, ist die erneute Ausführung kein Problem. Wenn es versucht den Scheduler zu verlassen, wird

es unweigerlich in die Methode `boost()` gelangen, wodurch die höherwertigen Objekte verarbeitet werden.

```
1:   void Instance::appear(){
2:       protect.retne();
3:       boost();
4:   }
```

Abbildung 3.48: Die `appear()` Methode eines aktiven Objektes

In der Situation des initialen Startens eines Objektes sieht die Sache aber etwas anders aus. Das Objekt verläßt das Planugswesen über einen anderen Weg. Nachdem die Methode `resume()` den Wechsel zum Kontext des Objektes vorgenommen hat, wird das Objekt über die `kickoff`-Funktion angesprungen. Dadurch wird das Objekt gemäß Abschnitt 3.3.7.3 gestartet. Wenn also keine Möglichkeit bestehen würde den kritischen Abschnitt zu verlassen und auch den Scheduler erneut zu betreten, kann dieses Objekt ungehindert rechnen, obwohl höherwertige Objekte auf der Liste stehen. Eine Prioritätsumkehr ist das Ergebnis, die solange bestehen würde, bis eine Unterbrechung oder das Objekt selber das Planungswesen betritt. Deshalb spielt die Methode `appear()` hier eine entscheidende Rolle. Wie das Verlassen des kritischen Abschnittes und das abermalige Eintreten in den Scheduler realisiert wurde, ist in der Abbildung 3.48 zu sehen. Wenn das Objekt seinen Startvorgang durchführt, verläßt es den kritischen Abschnitt durch den Aufruf von *protect.retne()*. Danach wird durch die Methode `boost()` das Planungswesen angeworfen, und die Propagation der höherwertigen Objekte kann erfolgen.

3.3.8 Das Idleobjekt

Ob in einem System ein Idleobjekt vorhanden sein muß oder nicht, hängt stark von der Anwendung ab. Verwenden die Objekte der Applikation die Methode des Blockierens nicht, braucht in dem System auf jeden Fall kein Idleobjekt vorhanden sein. Kann davon ausgegangen werden, daß auch immer mindestens ein rechenbereites Objekt existiert, ist dies die gleiche Situation. Problematisch wird es nur, wenn nicht mit Bestimmtheit vorausgesetzt werden kann, daß ein aktives Objekt da ist. Bei einem solchen Szenario ist es unbedingt erforderlich, ein aktives Objekt, das Idleobjekt, zu haben, zusätzlich zu den anderen aktiven Objekten, die die Anwendungsaufgaben erledigen. Wenn sich beispielsweise auch das letzte aktive Objekt blockieren möchte, wohin sollte es dann wechseln? Das Systemverhalten ist nicht definiert, wenn dieser Fall eintritt.

Die Erzeugung eines Idleobjektes ist mit wenig Aufwand verbunden. Ein solches Objekt ist ein gewöhnliches aktives Objekt. Folglich ist es von der Klasse *Instance* abgeleitet. Das objektspezifische Verhalten wird durch eine einfache Endlosschleife implementiert. Da beim erstmaligem Starten des Objektes automatisch der Scheduler betreten wird, muß dies nicht explizit getan werden.

Ist der Prozess in der Endlosschleife angekommen, ist das nicht problematisch, weil es sich bei dem System um ein voll präemptives handelt. So ist gewährleistet, daß eine Unterbrechung jederzeit verarbeitet werden kann. Beim Beenden des Interruptobjektes werden dann automatisch wieder alle anstehenden Objekte propagiert, wobei natürlich auch eine Umschaltung zu anderen aktiven Objekten möglich ist.

Kapitel 4

Ergebnisse

In dem folgenden Kapitel werden die erreichten Ergebnisse dargestellt und diskutiert. Dabei liegt ein Schwerpunkt bei den Systemgrössen, wobei beschrieben wird, wie diese zu bewerten und zu interpretieren sind. Ein weiterer Punkt stellt die Diskusion über mögliche Vor- und Nachteile von Entwurfsentscheidungen dar. Prinzipiell soll dabei beleuchtet werden, wie diese oder jene Entscheidungen sich nachteilig auf Eckdaten des entstandenen Systems auswirken.

4.1 Systemgrößen

Als Systemgrößen werden allgemein Werte bezeichnet, die die spezifischen Eckdaten des Systemes wiederspiegeln. Unter anderem kann hier der Speicherverbrauch oder die Laufzeit einzelner Komponenten oder des Gesamtsystemes betrachtet werden. Es wird der Resourcenverbrauch angezeigt. Weiterhin wird beleuchtet, ob spezielle Hardwarefähigkeiten benötigt werden.

Für den Bereich der tiefsten eingebetteten System spielen diese Daten eine wichtige Rolle, da anhand dieser entschieden werden kann, ob das erstellte System auf einer Zielplattform einsatzfähig ist.

Grundsätzlich kann zwischen zwei Versionen eines Systemes unterschieden werden. Die ersten Werte fallen in der Entwicklungs- und Testphase an. Dabei werden natürlich auch Fragmente im übersetzten System enthalten sein, die nicht Bestandteil der entgültigen Produktfassung sind. Zu diesen Artefakten zählen beispielsweise Debuginformationen sowie Testausgaben. Daher ergeben sich unterschiedliche Größen zwischen der *Develop*- und der *Product*versionen eines erzeugten Betriebssystems.

Die zum Teil enormen Größenunterschiede zwischen den Versionen erklären sich, wenn das Modell bekannt ist, nachdem der Übersetzungs- und Bindevorgang arbeitet. Hier sei nur kurz erklärt, daß während der Entwicklungsphase ein kompletes Objektmodul dazugebunden wird, sobald ein Bestandteil von diesem benötigt wird. Bei der *Product*version wird hingegen jedes Modul in kleinere Einheiten zerlegt. Diese werden dann getrennt übersetzt. Beim Binden werden dadurch nicht die vollständigen Bausteine gelinkt, sondern nur die referenzierten Teile zum Gesamtsystem zusammengefügt. Eine weiter

Möglichkeit in der *Product*version an Größe zu sparen ist der Einsatz von *inlining*. Dabei werden Funktion textlich an ihren Aufrufstellen ersetzt. Bei sehr kleinen Methoden sowie bei Routinen, die nur einmalig aufgerufen werden, ist diese Vorgehensweise sinnvoll. Wird dieser Mechanismus jedoch nicht mit Bedacht eingesetzt, kann er sich negativ auswirken, indem das System an Größe gewinnt. Ein Werkzeug, welches eine möglichst optimale *inlining*-Struktur errechnet, wäre ein gewaltiger Fortschritt.

In den folgenden Beschreibungen werden die Größen für die *Develop*- und die *Product*variante angegeben. Dadurch wird ersichtlich, wieviel Speicherplatz eingespart werden kann, wenn die *Product*version anstatt der Entwicklervariante generiert wird.

4.1.1 Das Testsystem

Die Systemgrößen wurden auf einem Linuxsystem Kernelversion 2.2.16 mit x86 Architektur gemessen. Die entstandene Klassenhierarchie wurde mit Hilfe des GCC Compilers egcs in der Version 2.91.66 übersetzt. Dabei wurden Messungen im *Develop*- und *Product*modus durchgeführt.

4.1.2 Die Listenverwaltung

Bei der Listenverarbeitung wurde nicht jede einzelne Methode für sich ausgemessen. In einem Testprogramm wurden die Methoden des Usher referenziert, wodurch sie gemeinsam in einer ausführbaren Anwendung vorlagen. Um an die Größen des Systems zu gelangen, wurde im weiteren Verlauf die PURE-Klassenbibliothek einmal im *Develop*- und ein weiteres mal im *Product*modus übersetzt.

Die Werte in der Tabelle 4.1 spiegeln die erreichten Ergebnisse wieder. Es ist ein deutlicher Unterschied zwischen der *Develop*- und der *Product*version erkennbar. Dieser bezieht sich nicht nur auf den Bereich des ausführbaren Codes, sondern auch auf die benötigten Daten.

Usher								
Übersetzung-	Code		Data		BSS		Total	
modus	Bytes	%	Bytes	%	Bytes	%	Bytes	%
Develop	2540	100	252	100	24	100	2816	100
Product	248	9,7	12	5,7	0	0	260	9,2

Tabelle 4.1: Code- und Datengrößen der Listenverwaltung *Usher*

Diese erheblichen Unterschiede ergeben sich durch die Tatsache, daß in der *Product*version keine Testausgaben und Debuginformationen mehr vorhanden sind. Desweiteren sind auch nur die tatsächlich benötigten Systembestandteile intergriert. Durch die verschiedenen Übersetzungsmodi wurden rund 90% des Codes, rund 94% der Daten und 100% der global zu initialisierenden Daten

eingespart. Das Gesamtsystem hat nur noch eine 9,2% Größe gegenüber der Entwicklerversion.

Die zwölf Byte Daten in der *Product*version setzen sich aus einem acht Byte großen Endelement und einem vier Byte Kopfzeiger zusammen. Für die Nutzung der Liste ist diese Grundausstattung an Daten ausreichend.

Werte in dieser Größenordnung sind sicherlich zufriedenstellend, wenn Bedacht wird, daß die Liste priorisierte Elemente aufnimmt und nebenläufige Vorgänge erkennt sowie entsprechend behandelt werden.

4.1.3 Der Scheduler

Bei dem Ausmessen des Schedulers wurde eine leere Listenimplementierung an das Planungswesen angeschlossen. Dadurch wurde es möglich den Scheduler beziehungsweise eigentlich nur den zentralen Algorithmus größenmäßig zu erfassen. Zu diesem Zwecke wurde ein kleines Testprogramm geschrieben, in dem sämtliche Funktionen des Planungsalgorithmus aufgerufen werden. Somit sind alles Methoden referenziert.

Da der Scheduler so konfiguriert werden kann, daß er in zwei unterschiedlichen Betreibsarten läuft, wurden insgesamt vier Messungen durchgeführt. Dabei wurden pro Konfiguartionsvariante zwei Messungen erbracht jeweils eine für den *Develop*- und den *Product*modus.

Scheduler							
Übersetzungs-		Code		Data		Total	
modus	*fame*Propagate...	Bytes	%	Bytes	%	Bytes	%
Develop		6957	100	704	100	7661	100
Product		737	10,5	4	0,5	741	9,8
Develop	gesetzt	6845	100	704	100	7549	100
Product	gesetzt	697	10,2	4	0,5	701	9,4

Tabelle 4.2: Code- und Datengrößen des Schedulers

In der Tabelle 4.2 können die gemessenen Werte betrachtet werden. Wie schon bei der Listenverwaltung, ist auch hier ein enormer Größenunterschied zwischen den verschiedenen Übersetzungsmodi zu beobachten. Hingegen unterscheiden sich die Konfigurationsvarianten des Schedulers nur unwesentlich von einander. Dies ist sicherlich auch nicht verwunderlich, da nur ein sehr kleiner Teil innerhalb des Planungsalgorithmus verändert wurde.

Die Daten, die der Scheduler benötigt, beschränken sich auf wenige Bytes. Die vier Datenbytes entfallen auf die globale Schutzvariable *protect*, welche vom Typ *Locker* ist. Somit hält sich der Verbrauch an RAM-Speicher sehr in Grenzen.

4.1.4 Das Gesamtsystem

Das Gesamtsystem setzt sich aus dem Scheduler und der Listenverwaltung zusammen. Da das entstandene System auf bereits bestehende Komponenten der PURE-Klassenbiblothek aufsetzt, mußte um ein repräsentatives Bild für Größen zu bekommen, diese Menge ersetzt werden. Würde nicht so vorgegangen, erfolgt das Ausmessen von Bestansteilen, die nicht im Zuge dieser Arbeit entwickelt wurden. Daher ist die Beschaffung des Stacks sowie die Basisklassen für aktive Objekte durch adäquate leere Implemente realisiert.

Wiederum wurden Messungen im *Develop*- und *Product*modus durchgeführt. Dabei fielen vier Messenreihen an, weil durch die Konfigurationsmöglichkeit innerhalb des Schedulers ein einmaliges Ausmessen der beiden Übersetzungsmodi nicht ausreichend ist. In der Tabelle 4.3 können die gesam-

Das Gesamtsystem (Scheduler plus Listenverwaltung)							
Übersetzungs-		Code		Data		Total	
modus	*famePropagate...*	Bytes	%	Bytes	%	Bytes	%
Develop		7309	100	716	100	8025	100
Product		1025	14	16	2,2	1041	12,9
Develop	gesetzt	7197	100	716	100	7913	100
Product	gesetzt	961	13,3	16	2,2	977	12,3

Tabelle 4.3: Code- und Datengrößen des Gesamtsystems

melten Werte betrachtet werden. Wie nicht anders zu erwarten, kann auch diesmal eine deutliche Größenabnahme von der *Develop*- zur *Product*fassung erreicht werden.

Auch der Datenaufwand des Gesamtsystems hält sich in Grenzen. Es werden lediglich sechzehn Bytes an Daten verwendet. Die schon beschriebenen zwölf Bytes der Listenverwaltung, die sich aus dem Listenkopfzeiger und dem Listenendelement zusammen setzen, sowie aus den vier Bytes des Schedulers für den Schutzmechanismus des *Locker*.

Kapitel 5

Diskussion und Ausblick

Das in dieser Arbeit entwickelte System, erweitert die PURE-Klassenbibliothek um ein präemptives Planungswesen für aktive und passive Objekte. Der Betreibssystemfamilie wurde ein neues Mitglied hinzugefügt. Ausgehend von der Problemstellung wurde ein System gebaut, welches auch den Anforderungen kleinster eingebetteter System gerecht wird.

Das System unterstützt das koordinierte Abarbeiten von Objekten mit Hilfe von Prioritäten. Dabei werden aktive und passive Objekte durch eine Instanz verwaltet. Der zentrale Algorithmus für die Propagation der verschidenen Objektvariationen sollte so wenig als möglich Unterbrechungen unterbinden, damit keine Ereignisse verlorenen gehen. Diese Aufgabe wurden erfolgreich gelöst, wobei ein System entstand, welches reaktionsfreudig ist, weil es sehr kurze kritische Abschnitte aufweist.

Für die Lösung der Aufgabe wurden zwei Möglichkeiten in Erwägunug gezogen, die im Kapitel 2 vorgestellt wurden. Da die Variante mit der gemeinsamen Verwaltung innerhalb einer Liste erfolgsversprechender war, wurde diese implementiert. Um herauszufinden, welche jedoch die optimalere Herangehensweise bezüglich der Systemgrößen darstellt, müßte auch die andere Varianten realisiert werden. Eine weitere Arbeit könnte sich mit der anderen Methode beschäftigen, wodurch eine Vergleichsmöglichkeit entstünde.

Die bei den durchgeführten Messungen ermittelten Werte, die größenordungsmäßig in den Bereich der tiefsten eingebetteten Systeme fallen, spiegeln sich die geringen Resourcenanforderungen des neuen PURE-Familienmitgliedes wieder. Daher ist es für den Einsatz in diesen Umgebungen prädestiniert. Prinzipiell ist bei den beschriebenen Systembestandteilen und deren Größen immer die gesamte Funktionalität der Komponete enthalten. Wird jedoch nur ein geringerer Funktionsumfang benötigt, kann zusätzlich ein wesentlicher Anteil an Codegröße eingespart werden.

Ein weiterer möglicher Entwicklungsschritt stellt die Erarbeitung einer Liste dar, die ohne eine hardwareseitige Unterstützung auskommt. Es laufen bereits Forschungsprojekte, die sich mit den Fragestellungen der nicht blockierenden Synchronisation beschäftigen, so daß möglicherweise in naher Zukunft Algorith-

men bereit stehen, die ohne spezielle Hardwareanweisungen auskommen.

Zu den Entwurfsentscheidung sei noch angemerkt, daß die speziellen Stacks nicht unbedingt nötig sind. Im Verlauf der Entwicklung stellte sich heraus, daß ein Umschalten der Kontext zwischen aktiven Objekten immer innerhalb eines kritischen Abschnittes erfolgt. Daher wird die Funktion des impliziten Umschaltens nicht notwenigerweise benötigt. Größeneinsparungen sind die Folge, wenn die Speicheranforderungsmethode für spezielle Stacks nicht gebraucht wird. Die Stacks werden dann entweder schon während der Generierungszeit des Systems ausgerichtet erzeugt oder es werden unausgerichtete Stack favorisiert, wobei dann aber ein zusätzlicher *life*-Pointer verwendet werden muß, über den der aktuell laufende Prozeß identifiziert wird.

Ein Problem entsteht bei der Dimensionierung der Stacks für die aktiven Objekte. Durch verschiedene Konfigurationen und die im hohen Maße angewandte Technick der *Inline*-Expansion können die Aufruffolgen von Systemfunktion enorm variieren. Da im System weiterhin noch kaskadierte Unterbrechungen zugelassen sind, kann die benötigte Menge an Stapelspeicher nicht sicher vorausgesagt werden. Ein Entwicklungswerkzeug zur Berechnung dieser Größe wäre sehr sinnvoll. Eine weitere Arbeit könnte sich mit diesem Thema auseinandersetzen.

Durch die Anwendung von *Inline*-Expansion kann es vorkommen das Funktion, die als klein angenommen werden, im System an ihren zahlreichen Aufrufstellen textlich ersetzt werden, wodurch es zu einer Codeaufblähung kommt. Ein Wergzeug welches die optimale Ersetzungsstruktur berechnet, wäre ein großes Hilfsmittel für die Verkleinerung der Systemgrößen. Ein Ergebnis eines Projekt auf diesem Gebiet ist sicherlich ein nützliches Tool für die Softwareentwicklung.

Literaturverzeichnis

[1] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley, 1998. ISBN 0-2011-33143-8.

[2] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999. ISBN 0-201-82467-1.

[3] Ralf Guido Herrtwich and Günter Hommel. *Kooperation und Konkurrenz*. Studienreihe Informatik. Springer Verlag, 1989.

[4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[5] Samual J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1989.

[6] Scott Mayers. *Effektiv C++ programmieren*. Addison-Wesley, 1992. erste Auflage.

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

[8] Claus Schirmer. *Die Programmiersprache C*. Hanser, 1992. dritte Auflage.

[9] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 9.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, Paderborn, 1998.

[10] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 270–277, Newport Beach, California, March 15–17, 2000. IEEE Computer Society. ISBN 0-7695-0607-0.

[11] W. Schröder-Preikschat. PEACE—The Evolution of a Parallel Operating System. Arbeitspapiere der GMD 646, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany, May 1992.

[12] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[13] Helge Sichting. Nichtblockierende synchronisierte segmentorientierte Haldenverwaltung. 2000.

[14] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997. third edition.