Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Leibniz
Universität
Hannover

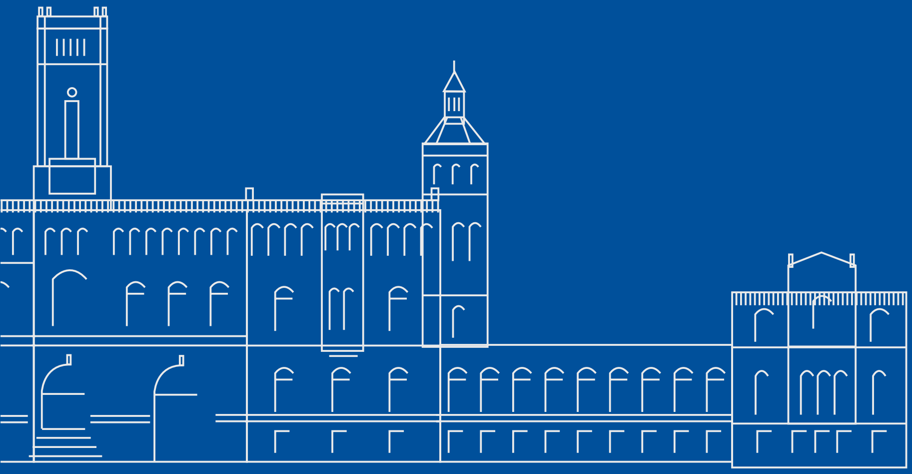
Malte Bargholz

InterSloth: Globales hardware-gesteuertes Scheduling in einem Multikern-Echtzeitbetriebssystem auf RISC-V

Bachelorarbeit im Fach Technische Informatik

8. August 2018

Please cite as:
Malte Bargholz, “InterSloth: Globales hardware-gesteuertes Scheduling in einem Multikern-Echtzeitbetriebssystem auf RISC-V” Bachelor’s Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, August 2018.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

InterSloth: Globales hardware-gesteuertes Scheduling in einem Multikern-Echtzeitbetriebssystem auf RISC-V

Bachelorarbeit im Fach Technische Informatik

vorgelegt von

Malte Bargholz

geb. am 23. April 1995
in Dannenberg

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Bernardo Wagner**
Betreuer:	**M.Sc. Gerion Entrup** **M.Sc. Christian Dietrich**

Beginn der Arbeit:	**09. April 2018**
Abgabe der Arbeit:	**09. August 2018**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Malte Bargholz)
Hannover, 8. August 2018

ABSTRACT

Partitioned operating systems often suffer from a suboptimal utilization. While global operating systems dynamically assign waiting threads to processor cores, partitioned operating systems statically assign threads to cores at compile-time. The optimal assignment usually depends on environmental conditions and is therefore hard to determine during the system development period. Suboptimal assignment leads to under- or overutilization of individual processor cores and therefore to a suboptimal system utilization.

An additional problem in operating systems is the separated priority space between application threads and interrupt service routines. Even low-priority interrupts always take precedence before high-priority threads, leading to the problem of priority inversion. This is especially detrimental for real-time operating systems, where a certain response time needs to be guaranteed. The Sloth-family of real-time operating systems solves this problem by mapping threads to interrupt service routines unifying the priority space in the process. All scheduling is moved to the interrupt controller, simplifying the operating system implementation and increasing task switching performance. Later work also implemented a multi core variant of the Sloth operating system (MultiSloth), which uses a partitioned scheduling approach.

This thesis implements a MultiSloth variant with global scheduling to increase the system utilization of multicore Sloth-systems. To achieve this the interrupt controller needs to be modified, as current interrupt controllers do not offer required functionality for interrupt migration and strict priority obedience. An implementation of InterSloth on the open instruction-set architecture RISC-V shows promising results needing only 160 cycles to activate a task on a remote core.

KURZFASSUNG

Partitionierte Multiprozessor-Betriebssysteme leiden häufig unter einer suboptimalen Auslastung. Im Gegensatz zu globalen Betriebssystemen, werden bei ihnen Programme fest an Prozessorkerne gebunden und zur Laufzeit nur auf dem zugewiesenen Prozessorkern ausgeführt. Die Zuteilung von Programmen zu Prozessorkernen ist dabei häufig von der Systeminteraktion mit seiner Umwelt abhängig und daher zur Entwicklungszeit nur schwer optimal feststellbar. Hierdurch kann es zu einer ungleichmäßigen Auslastung der Prozessorkerne und damit zu einer geringeren Gesamtsystemauslastung kommen.

Ein weiteres Problem in Betriebssystemen ist der getrennte Prioritätenraum zwischen Unterbrechungsbehandlungsroutinen und Anwendungsfäden. So kann es durch die asymmetrische Natur der Unterbrechungsbehandlung dazu kommen, dass eine niederpriorere Unterbrechungsbehandlungsroutine einen hochprioren Anwendungsfaden unterbricht. Vor allem in Echtzeitbetriebssystemen kann dies zur Verletzung der Echtzeitgarantien führen. Die Sloth-Betriebssystemfamilie löst dieses Problem der Prioritätsinversion, durch eine Auslagerung der Fadeneinplanung in den Unterbrechungskontroller. Zusätzlich zeigt Sloth noch weitere nicht-funktionale Eigenschaften wie eine geringe Kerngröße und Einlastungslatenz. In folgenden Arbeiten wurde das Sloth-Echtzeitbetriebssystem, welches als Uniprozessorbetriebssystem entworfen wurde, um eine partitionierte Multiprozessorunterstützung erweitert.

Um auch im Multiprozessorfall eine gute Auslastung in Sloth zu erreichen, soll dieses nun im Rahmen dieses Projekts um eine globalen Fadeneinplanung erweitert werden. Hierfür muss zunächst der Unterbrechungskontroller um eine Unterstützung von Unterbrechungsmigration und strikte Prioritätstreue erweitert werden. Auf der offenen RISC-V Architektur ergibt sich für das in dieser Arbeit behandelte InterSloth-Betriebssystem eine Latenz von 160 Takten zur Aktivierung eines Applikationsfadens.

INHALTSVERZEICHNIS

1

EINLEITUNG

Bei der Einlastungsplanung von Multiprozessorsystemen wird zwischen zwei verschiedenen Ansätzen unterschieden. Eine partitionierte Einlastungsplanung teilt jedem Programm zur Übersetzungszeit einen festen Prozessorkern zu, wohingegen eine globale Einlastungsplanung dynamisch zur Laufzeit einen Prozessorkern auswählt.

Findet eine suboptimale Zuteilung der Programme an Prozessorkerne statt oder wird das reale Auslastungsszenario falsch eingeschätzt, so kann es zu einer stark unausgewogenen Auslastung bei partitionierten Multiprozessorsystemen führen. Hierdurch sinkt die Gesamtauslastung und die benötigten Hardwareanforderungen für eine gegebene Anwendung steigen.

Systeme mit einer globalen Einlastungsstrategie besitzen dieses Problem nicht. Programme werden dynamisch Prozessorkernen zugeordnet und eventuell verdrängte Programme können auf einen anderen Kern migriert werden, um so eine bessere Auslastung des Systems zu erreichen.

Auch im Bereich der Echtzeitanwendungen finden Multiprozessorsysteme, insbesondere aufgrund der größeren Energieeffizienz, immer mehr Anwendung. So wurde auch das Sloth-Echtzeitbetriebsssystem, welches zunächst nur als Uniprozessorvariante existierte, unter dem Namen MultiSloth für Multiprozessorsysteme angepasst. Allerdings implementiert MultiSloth einen strikt-partitionierten Einlastungsplanungsansatz und leidet somit unter den oben beschriebenen Auslastungsproblemen.

Die Sloth-Betriebssystemfamilie entstand im Gedanken, dass eine tiefere Anpassung des Betriebssystems an die unterliegene Hardware zu Geschwindigkeitsverbesserungen führen könnte. Die Besonderheit der Sloth-Betriebssystemfamilie ist die Nutzung des Unterbrechungskontrollers zur Einlastungsplanung. Programme werden als Unterbrechungsroutinen modelliert und können somit gleichartig priorisiert werden wie Unterbrechungen. Auf diese Weise kann das häufig auftretende Problem der Prioritätsinversion, bei dem eine niedrigpriore Unterbrechungsbehandlungsroutine ein hochpriores Programm verdrängen kann, effizient verhindert werden. Weiterhin wird so die Größe und Komplexität des Betriebssystems stark verringert und erlaubt eine einfachere Verifizierung.

Mit InterSloth soll eine Variante des Sloth-Betriebssystems geschaffen werden, welche die vorteilhaften nicht-funktionalen Eigenschaften des Sloth-Betriebssystems erhält, aber eine globalen Einlastungsplanungsansatz implementiert, um auf diese Weise die Gesamtauslastung des Systems zu verbessern.

Eine Evaluation der vorhandenen Unterbrechungskontroller ergab jedoch zunächst, dass keiner der kommerziell verfügbaren Unterbrechungskontroller eine Migration von Unterbrechung erlaubt, wodurch die Implementierung eines globalen Einlastungsplanungsansatzes unmöglich ist [Bew16]. So wurde durch Christian Bewermeyer ein Unterbrechungskontroller entworfen, welcher neben einer strikten Prioritätstreue ebenfalls die Möglichkeit zur Migration, d.h. Neuzustellung, bereits in Behandlung befindlicher Unterbrechungen erlaubt. Im Rahmen dieser Arbeit, soll zunächst eine Portierung dieser Funktionalität in den Unterbrechungskontroller der quelloffenen Architektur RISC-V erfolgen. Hierfür wird der bekannte Systememulator QEMU so erweitert, dass er das gewünschte Verhalten zeigt. Anschließend wird das Sloth-Betriebssystems auf die RISC-V Architektur portiert, um schlussendlich eine globalen Einlastungsplanung im Sloth-Betriebssystem zu realisieren. Der Rest dieser Arbeit ist wie folgt strukturiert: Kapitel 2 führt zunächst die Sloth-Betriebssystemfamilie ein und erklärt dann die Zielplattform RISC-V und den von Christian Bewermeyer entwickelten Unterbrechungskontroller MIRQ-V. Anschließend geht Kapitel 3 auf die genaue Anpassung des RISC-V Unterbrechungskontroller, sowie die Architektur von InterSloth ein. Als letztes wird dieses dann in Kapitel 4 in Bezug auf Korrektheit und Geschwindigkeit evaluiert wird.

GRUNDLAGEN 2

Das Sloth-Projekt, aus dem diese Arbeit hervorgeht, verwendet die weitverbreitete OSEK-OS Semantik, welche im Folgenden zuerst erklärt werden soll. Anschließend wird das Sloth-Projekt eingeführt und auf seine Erweiterungen, besonders die Mehrkernvariante MultiSloth eingegangen. Weiterhin wird der *MIRQ-V* Unterbrechungskontroller [Bew16] erklärt, welcher die Grundlage für die Modifizierung des PLIC der RISC-V-Architektur bildet. Als Letztes wird die RISC-V Architektur im Detail mit Fokus auf das Unterbrechungssubsystem erläutert.

2.1 OSEK-OS

OSEK [OSE05] ist ein in 1993 gegründetes Gremium zur Standardisierung von „ **O**ffenen Systemen und deren **S**chnittstellen für **E**lektronik im **K**raftfahrzeug“ welches vor allem aus Automobilherstellern besteht. Seine bedeutendste Entwicklung, das sogenannte OSEK-OS, entstand nach dem Zusammenschluss mit der 1988 gegründeten französischen VDX Initiative im Jahre 1994. OSEK-OS ist daher, seit seiner Veröffentlichung im Jahre 1995, unter dem Namen OSEK oder OSEK/VDX bekannt. Nach mehreren Revisionen wurde es 2005 unter dem ISO-Standard 17356-3 veröffentlicht. Aktuell wird es unter dem AUTOSAR Standard weiterentwickelt.

Es gibt zwei Varianten von OSEK. Zunächst das nur unter OSEK bekannte ereignisgesteuertes System, und ein unter OSEKtime bekanntes zeitgesteuertes System. Da es sich bei dem Sloth-Projekt um ein ereignisgesteuertes System handelt, soll an dieser Stelle nur auf die ereignisgesteuerte OSEK-Variante eingegangen werden (siehe Abschnitt 2.2).

Bei OSEK handelt es sich um ein statisches Betriebssystem, d.h. zur Laufzeit ist keine Veränderung der Konfiguration möglich. Sämtliche Betriebsmittelreservierungen und Zuweisungen sowie Programmabläufe müssen zur Übersetzungszeit bekannt sein. OSEK unterscheidet zwischen zwei verschiedenen Arten von Prozessen: Unterbrechungsroutinen (OSEK Interrupt-Service Routine (ISR)) und normalen Programmen (OSEK *Tasks*). Es existieren zwei verschiedenen Varianten von ISRs. Typ-1 ISRs dürfen keine Betriebssystemfunktionen aufrufen und müssen somit nicht mit dem Betriebssystem synchronisiert werden, wohin gegen es Typ-2 ISRs erlaubt ist Betriebssystemfunktionen aufzurufen. Ein OSEK-Task existiert ebenfalls in zwei verschiedenen Varianten (*Basic* und *Extended*) und ist die Abstraktion für ein in sich abgeschlossenes Teilprogramm innerhalb des Gesamtprogramms. Er kann durch andere Tasks oder das Betriebssystem aktiviert werden, woraufhin eine Instanz des Teilprogramms ausgeführt wird.

Die Verwaltung aller aktiven Tasks unterliegt dem Betriebssystemeinplaner. Er entscheidet anhand der Taskpriorität, welcher Task als nächstes auf der CPU ausgeführt wird. Die Priorität der ISRs ist dabei implizit höher, d.h. eine ISR verdrängt immer einen Task, aber nicht umgekehrt. Zusätzlich existiert noch eine virtuelle Prioritätsebene für den Betriebssystemeinplaner, welche über der maxi-

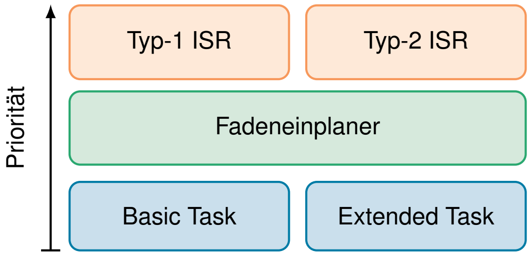


Abbildung 2.1 – OSEK-Prioritätsebenen

malen Taskpriorität liegt, um die Verwaltung der Tasks zu ermöglichen (siehe Abbildung 2.1).
Je nach Zeitpunkt der Einplanungsentscheidung spricht man von *full preemptive*, *mixed preemptive* oder *non preemptive* Einplanung.
Wie der Name schon vermuten lässt, handelt es sich bei *full preemptive*-Einplanung um ein verdrängendes Verfahren, d.h. ein Task kann zu jedem Zeitpunkt von höherprioren Tasks verdrängt werden. Im Gegensatz dazu kann ein Task bei *non preemptive*-Einplanung lediglich bei dem Aufruf von bestimmten Systemfunktionen verdrängt werden. Dazu gehören Systemfunktionen zur Terminierung eines Tasks, Warten auf ein Ereignis und ein direkter Aufruf des Einplaners. Bei *mixed preemptive* Einplanung ist die Entscheidung, ob ein Task jederzeit verdrängbar ist, abhängig von der Taskkonfiguration und kann für jeden Task einzeln konfiguriert werden.
Wie zuvor beschrieben existieren zwei Arten von Tasks in OSEK: *Basic Tasks* und *Extended Tasks*. *Basic Tasks* besitzen eine run-to-completion Semantik und können nur durch ISRs oder das Betriebssystem verdrängt werden. *Extended Tasks* hingegen können mit Hilfe des Systemaufrufes `WaitEvent()` zusätzlich auf bestimmte Ereignisse warten und sind somit abhängig von anderen Tasks. Ist das gewünschte Ereignis bei dem Aufruf von `WaitEvent()` nicht vorhanden, so blockiert der Task und ein niederpriorer Task kann vom Betriebssystem ausgeführt werden. Erst durch einen Aufruf des zugehörigen Systemaufrufes `SetEvent()` kann ein auf diese Weise blockierter Task wieder als ausführbar markiert werden und für die Einplanungsentscheidung berücksichtigt werden.
Nach der Aktivierung durch einen Aufruf von `ActivateTask()` läuft eine Instanz eines Tasks solange, bis er entweder durch einen höherprioren Task verdrängt wird, er sich selbst durch einen Aufruf von `TerminateTask()` beendet oder im Falle von *Extended Tasks* an einem `WaitEvent()` Aufruf blockiert. Eine externe Terminierung von Tasks ist somit nicht möglich. Neben Terminierung von sich selbst durch einen `TerminateTask()` Aufruf bietet OSEK ebenfalls die Möglichkeit des *Task Chainings*. Hierbei ruft der sich beendende Task anstelle von `TerminateTask()` die Funktion `ChainTask()` mit dem zu startenden Task als Parameter auf. Der aufrufende Task wird beendet und gleich im Anschluss wird der an `ChainTask()` übergebene Task vom Betriebssystem aktiviert.
Neben Tasks besitzt OSEK zur Abstraktion zeitlich abhängiger Ereignisse sogenannte Alarme. Diese können entweder wiederkehrend oder einmalig sein und müssen ebenfalls zur Übersetzungszeit definiert und bekannt sein. Es gibt so zeitrelative als auch zeitabsolute Alarme, welche sich beide auf einen Implementationsspezifischen Zähler beziehen. Zur Behandlung von Alarmereignissen werden sogenannte *Alarm Callbacks* definiert, welche analog zu Typ-1 und Typ-2 ISRs auf einer höheren Prioritätsebene als normale Tasks ausgeführt werden.
Zur Koordinierung von verschiedenen Tasks besitzt OSEK-Ressourcen. Jede Ressource kann nur von einem Task gleichzeitig besessen werden und wird durch die Systemaufrufe `GetResource()` bzw. `ReleaseResource()` in Besitz genommen bzw. freigegeben.
OSEK garantiert zusätzlich, dass bei der Inbesitznahme einer Ressource kein Deadlock oder Prioritätsinversion auftritt.

Eine Prioritätsinversion ist dabei das Phänomen, dass ein niedrigpriorer Task durch Inbesitznahme einer geteilten Ressource einen höherprioren Task, welcher ebenfalls diese Ressource anfordert, in seiner Ausführung blockieren kann.
Zur Verhinderung von Deadlocks und Prioritätsinversion kommt in OSEK das sogenannte Priority Ceiling Protocol (PCP) zum Einsatz. Hierfür wird jeder geteilten Ressource (R_i) zur Übersetzungszeit eine Priorität abhängig von allen Tasks zugeordnet nach der Formel:

$$\text{Priorität}(R_i) = \max(\text{Priorität}(T_i) \mid \forall T_i \in \text{ Tasks } \wedge T_i \text{ benutzt } R_i) \tag{2.1}$$

Eine Ressource R_i erhält also die Priorität des Tasks, welcher unter allen Tasks, die R_i besitzen können, die maximale Priorität besitzt.
Nimmt ein Task eine Ressource in Besitz, so wird seine Priorität dynamisch auf die der Ressource zugeordnete Priorität angehoben und die vorherige Priorität gesichert. Während ein Task eine Ressource besitzt, kann er somit nicht von einem anderen Task, welcher ebenfalls diese Ressource anfordern könnte, verdrängt werden. Ansonsten wäre die Priorität des verdrängenden Tasks größer als die der Ressource zugeordnete Priorität, was im Widerspruch zu der Berechnungsvorschrift für Ressourcenprioritäten steht. Bei dem Freigeben der Ressource wird anschließend die Taskpriorität wieder auf den zuvor gesicherten Wert zurückgesetzt.
OSEK existiert in vier verschiedenen Konformitätsklassen, BCC1 und BCC2 sowie ECC1 und ECC2:

BCC1 Beschreibt die Basisklasse und beschränkt OSEK auf *Basic Tasks*, maximal ein Task pro Prioritätenlevel und maximal eine parallele Aktivierung eines Tasks.

BCC2 Enthält BCC1 und fügt mehrere Tasks pro Prioritätenlevel, eigene OSEK-Ressourcen (neben der virtuellen Einplaner Ressource) und mehrere parallele Aktivierungen von Tasks hinzu.

ECC1/ECC2 Verhalten sich wie BCC1/BCC2 und erlauben zusätzlich *Extended Tasks*.

Es sei an dieser Stelle bemerkt, dass *Extended Tasks* jedoch – unabhängig von der Konformitätsklasse – niemals mehrmals parallel aktiviert werden können.

2.2 Sloth-Betriebssysteme

Im Folgenden soll die Sloth-Betriebssystemfamilie von Echtzeitbetriebssystemen eingeführt werden, welche aus der Idee entstanden, dass Echtzeitbetriebssysteme effizienter sein könnten, wenn sie besser auf die darunterliegende Hardware zugeschnitten sind. Nach der Vorstellung des ersten Prototypen von Sloth im Jahre 2009, welcher zunächst nur einen Prozessor und *run-to-completion* Tasks unterstützte, wurde Sloth auf mehr als fünf Plattformen portiert und um blockierende Tasks (siehe Abschnitt 2.2.3) und eine Mehrkernprozessorunterstützung (siehe Abschnitt 2.2.4) erweitert.

2.2.1 Sloth

Traditionelle (Echtzeit)Betriebssysteme, wie OSEK-OS [OSE05] oder CiAO [Loh+09], unterscheiden zwischen zwei verschiedenen Arten von Kontrollflüssen. Während Unterbrechungen immer Vorrang haben und durch einen externen Unterbrechungskontroller priorisiert und verwaltet werden, werden Applikationsfäden durch das Betriebssystem verwaltet und priorisiert. Diese Aufteilung kann in Betriebssystemen zu dem Problem der Prioritätsinversion führen: Eine Unterbrechung mit einer niedrigen Priorität verdrängt ein Applikationsfaden mit hoher Priorität.
Um dieses Problem zu lösen, verfolgt das Sloth-Projekt [Hof+09] der Friedrich-Alexander Universität

Erlangen/Nürnberg den Ansatz, Applikationsfäden als Unterbrechungsbehandlungsroutinen zu implementieren. Ein geteilter Prioritätenraum entsteht, in welchem dem Unterbrechungskontroller bekannt ist, ob der gerade laufende Applikationsfaden eine höhere Priorität besitzt als die zuzustellende Unterbrechung. Prioritätsinversionen werden auf diese Weise implizit erkannt und verhindert. Des Weiteren wird durch die gleichartige Behandlung von Unterbrechungen und Applikationsfäden der sonst üblicherweise in Software implementierte Einplanungsalgorithmus nun in Hardware ausgeführt und ist damit signifikant schneller umsetzbar. Durch die Ausnutzung des Unterbrechungskontroller wird zudem nicht nur die Geschwindigkeit von Kontextwechseln, sondern auch ihr zeitlicher Determinismus erhöht. Zusätzlich wird die Komplexität des Betriebssystems reduziert. Dies erhöht nicht nur seine Wartbarkeit und öffnet Möglichkeiten zur formalen Verifikation des Betriebssystems, sondern reduziert auch die Größe des kompletten Systems drastisch. So entspricht ein OSEK-BCC1 kompatibles, generiertes Sloth-System mit einem Task, Alarm und einer Ressource nach der statische Konfiguration etwa 200 Zeilen Code oder etwa 800 Bytes kompiliert (beides ohne Bootcode).
Um die Einplanung von Anwendungsfäden in den Unterbrechungskontroller auszulagern stellt Sloth Anforderung an die Hardwareplattform:

- Der Unterbrechungkontroller implementiert die gewünschte Einplanungsstrategie.
- Der Unterbrechungskontroller unterstützt das Auslösen von Unterbrechungen aus Software.
- Es existieren so viele freie Unterbrechungsquellen (nicht mit externen Geräten verschaltet) wie es Applikationsfäden in der Zielanwendung gibt.
- Es existieren ausreichend freie Unterbrechungsprioritätenlevel, um sowohl Unterbrechungs- als auch Applikationsprioritäten abzubilden.

Um die Ergebnisse möglichst vergleichbar zu machen, wurde sich hierbei entschieden, ein zum *OSEK-OS* Standard (siehe Abschnitt 2.1) kompatibles Betriebssystem in Form eines reinen Uniprozessor Systems zu implementieren. Zudem reduziert eine Implementierung eines OSEK-kompatiblen

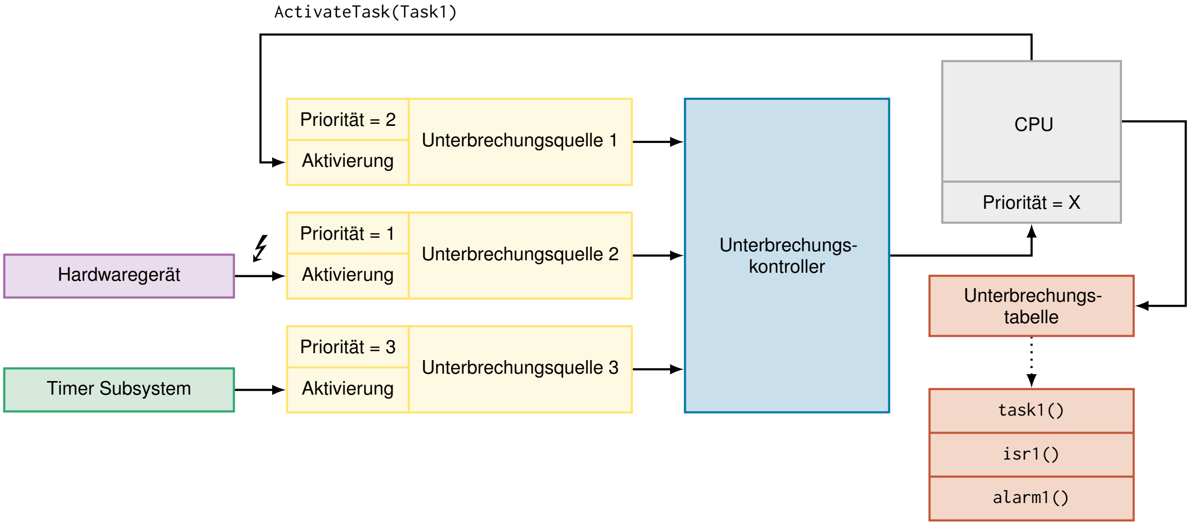


Abbildung 2.2 – Sloth-Struktur, adaptiert aus [Hof+09]

Betriebssystems den Portierungsaufwand bestehender Applikationen.

Ebenso wie OSEK, handelt es sich bei Sloth um ein statisches Betriebssystem, d.h. eine Änderung der Anwendungskonfiguration zur Laufzeit ist nicht möglich. Aus der Applikations- und Systemkonfiguration wird vor dem Übersetzen ein Betriebssystem generiert, welches exakt auf die Anforderungen zugeschnitten ist (siehe Abschnitt 2.2.2).

Applikationsfäden bzw. OSEK-Tasks sowie Typ-2 ISRs und Alarme werden statisch zur Übersetzungszeit freie Unterbrechungsquellen zugewiesen, welche dann bei dem Systemstart entsprechend ihrer Taskpriorität initialisiert werden. Um Alarmfunktionalität zu realisieren, werden anschließend die Unterbrechungsquellen, welche Alarme realisieren, mit dem Timersubsystem verbunden und dieses entsprechend der Alarmeigenschaften passend konfiguriert. Lediglich Typ-1 ISRs werden aufgrund ihrer Unabhängigkeit von Systemaufrufen getrennt behandelt und erhalten eine Priorität, welche höher ist als die aller Tasks, Typ-2 ISRs und Alarme. Task und Unterbrechung seien somit im Folgenden äquivalent und wenn von Tasks gesprochen wird, so seien stets auch Typ-2 ISRs bzw. Alarme gemeint.

Nach der Konfiguration der Unterbrechungsquellen wird für jeden Task ein Eintrag in der Unterbrechungstabelle erstellt. Wird die zum Task zugehörige Unterbrechung ausgelöst, so wird dieser dadurch automatisch aktiviert und ausgeführt.

Die Aktivierung eines Tasks kann sowohl aus der Software ausgelöst werden als auch durch eine externe Unterbrechung oder einen Timer. Abbildung 2.2 gibt einen Überblick über die Struktur des Sloth-Betriebssystems. Sämtliche Einplanungsentscheidungen werden im Unterbrechungskontroller gefällt, was zu einer deutlichen Reduzierung der Betriebssystemgröße führt. Dieser überprüft bei Aktivierung eines Tasks zunächst, ob die aktuelle CPU-Priorität niedriger ist als die des ausgelösten Tasks. Nur wenn dies der Fall ist, wird die CPU tatsächlich unterbrochen und führt mit Hilfe der zuvor eingerichteten Unterbrechungstabelle den neu aktivierten Task aus. So kann effizient verhindert werden, dass eine Prioritätsinversion auftritt. Sämtliche dafür benötigte Information wird bei Systemstart an den Unterbrechungskontroller übergeben und muss nicht im Betriebssystem verwaltet werden. Im Regelfall entspricht die aktuelle CPU-Priorität der des ausgeführten Tasks, jedoch wird sie auch zur System- und Ressourcensynchronisierung genutzt.

Die Terminierung eines Tasks entspricht im Sloth-Design einer einfachen *Return from Interrupt* Instruktion, welche den Unterbrechungkontroller anweist, weitere noch unbehandelte Unterbrechungen erneut auf Zustellbarkeit zu prüfen. Da bei der im Sloth-Projekt implementierten OSEK-Konformitätsklasse BCC1 nur *Basic Tasks* auftreten können, welche *run-to-completion* Semantik besitzen, ist vor der Ausführung des Tasks kein Stapelwechsel bzw. Kontextwechsel nötig. Taskaktivierungen können zwar beliebig gestapelt werden, jedoch wird niemals ein zuvor unterbrochener Task wieder ausgeführt, bevor der höherpriore unterbrechende Task zu Ende ausgeführt wurde. Die Aktivierung eines Tasks ist somit besonders effizient, da hierfür niemals Betriebssystemcode ausgeführt werden muss. Bei der Implementierung muss außerdem darauf geachtet werden, dass es sich bei dem `ActivateTask()` Systemaufruf laut Spezifikation um einen *synchronen* Systemaufruf handelt. Nach dem Auslösen einer Unterbrechung dauert es jedoch einige Zeit bis der Unterbrechungskontroller diese zustellt. Es kann somit passieren, dass zunächst für einige Takte weiterhin ein niederpriorer Task ausgeführt wird, obwohl zuvor ein höherpriorer Task aktiviert wurde. Dies ist ein Widerspruch zur Spezifikation. Abhängig von der Hardware muss daher eine gewisse Anzahl an Takten gewartet werden bevor sichergestellt werden kann, dass bei dem Verlassen des Betriebssystemsaufrufes sofort der richtige Task ausgeführt wird.

Zusätzlich zu *Basic Tasks* implementiert das Sloth-Projekt ebenfalls OSEK-Ressourcen. Die Implementierung orientiert sich an dem in OSEK definierten Priority Ceiling Protocol (siehe Abschnitt 2.1). Nimmt der aktuell laufende Task eine Ressource in Besitz, so wird die CPU-Priorität auf die zuvor errechnete Ressourcepriorität (siehe Gleichung (2.1)) angehoben. Wird anschließend ein anderer

Task aktiviert, so kann dieser den aktuellen Task nur unterbrechen, wenn er niemals auf die geteilte Ressource zugreift, da ansonsten seine Priorität immer niedriger ist als die Priorität die der Ressource zugewiesen wurde.

In der Geschwindigkeitsevalution des Sloth-Betriebssystem zeigt sich, dass durch Vereinigung von Applikationsfäden und Unterbrechungsbehandlungsroutinen ein signifikanter Geschwindigkeitsgewinn erzielt werden kann. So ist eine Taskaktivierung mit anschließender Ausführung und Kontextwechsel in 1200 ns bis 1480 ns auf einem 50 MHz Prozessor möglich. Das bereits erwähnte und ebenfalls stark optimierten CiAO Betriebssystem [Loh+09] benötigt für dieselbe Operation im Durchschnitt 4120 ns, was einer Verbesserung um den Faktor 2.8 entspricht. Insgesamt ergibt sich für verschiedene Einplanungsoperationen eine Verbesserung zwischen 1.0 und 2.8.

Zusammenfassend lässt sich sagen, dass ein Sloth-Betriebssystem aufgrund seiner Größe und Geschwindigkeit vorzuziehen ist, sofern die Zielplattform die notwendigen Bedingungen an den Unterbrechungskontroller erfüllt. Um allerdings zu den Erweiterungen des BCC1 Standards kompatibel zu sein, fehlt dem Sloth-Projekt die Möglichkeit für *Extended Tasks*, d.h. Tasks die aufgrund von Ereignissen blockieren können. Diese Problematik wurde in der Erweiterung *SleepySloth: Threads as Interrupts as Threads* behandelt (siehe Abschnitt 2.2.3). Bevor jedoch auf Sloth-Erweiterungen eingegangen wird, soll zunächst die Generatorstruktur, welche Sloth ermöglicht, eingegangen werden.

2.2.2 Sloth-Generator

Bei dem Sloth-Betriebssystem handelt es sich um ein generiertes Betriebssystem. Es wird abhängig von der Applikations- und Systemkonfiguration vor dem Übersetzen erstellt und ist daher speziell auf den Anwendungsfall zugeschnitten. Abbildung 2.3 zeigt die grundlegende Struktur des in Perl geschriebenen Sloth-Generators.

Zunächst wird der Anwendungscode und die zugehörige Systemkonfiguration, d.h. Taskprioritäten und Ressourcen, statisch analysiert. Hierbei wird unter Anderem einen flusssensitive Abhängigkeitsanalyse durchgeführt, welche danach vom Systemgenerator genutzt werden kann. Weiterhin wird sämtlichen Ressourcen an dieser Stelle eine Priorität nach Gleichung (2.1) (Priority Ceiling Protocol siehe Abschnitt 2.1) zugewiesen. Anschließend findet an dieser Stelle eine Verifikation der Systemkonfiguration statt, welche z.B. Konfigurationskonflikte entdeckt und dem Anwender entsprechend mitteilt. Das System ist nun vollständig konfiguriert und kann an den Systemgenerator übergeben werden. Dieser besteht aus zwei Teilen:

- Einem architekturspezifischen Teil, welcher durch das Architektur-Backend bereitgestellt wird.
- Einer generischen Sloth-Implementation.

Intern verwendet die Sloth-Implementation architekturspezifische Funktionen, welche durch die verschiedenen Plattformimplementationen (hier ARM, Infineon und Intel x86) implementiert werden. Die generischen Sloth-Implementation stellt somit einen Adapter zwischen den architekturspezifischen Funktionen und dem OSEK-Interface da, auch wenn das Interface zwischen Sloth und dem Architektur-Backend sich stark am OSEK-Interface orientiert. Weiterhin stellt das Architektur-Backend auch weitere architekturspezifische Teile des Betriebssystem wie Bootcode und Linkerskripte zur Verfügung.

Die Auswahl der Implementation geschieht hierbei durch den Anwender. Der Sloth-Generator besitzt, vorausgesetzt die gewünschten Architekturen unterstützen alle Sloth-Features, ebenfalls die Möglichkeit, ein System für verschiedene Architekturen zu generieren.

Neben der Anpassung des generischen Sloth-System durch architekturspezifische Funktion bietet der Sloth-Generator den Architektur-Backends ebenfalls die Möglichkeit, Code in Abhängigkeit der Systemkonfiguration zu generieren. Dies ist nötig, da zum Beispiel der Initialisierungscode des Unterbrechungskontrollers in den meisten Fällen abhängig von der Systemkonfiguration ist. Diese Mechanik wird ebenfalls zur Generierung der Task-Prologe in SleepySloth genutzt (siehe Abschnitt 2.2.3).

Der so generierte Code wird anschließend an den plattformspezifischen Compiler übergeben. Fast alle der Sloth-Systemaufrufe werden inzeilig (*inline*) kompiliert, wodurch praktisch das gesamte System direkt in den Anwendungscode integriert wird und somit besser optimiert werden kann. Nur so kann die kompetitive Größe und Geschwindigkeit von generierten Sloth-Systemen erzielt werden.

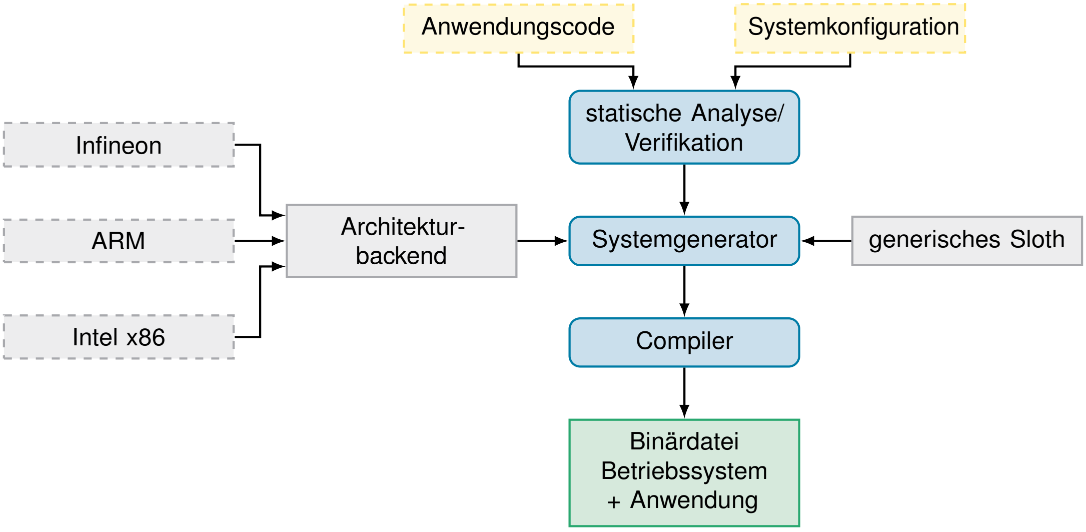


Abbildung 2.3 – Struktur des Sloth-Generators. Gelb dargestellt sind Eingabedaten, welche durch den Endanwender erstellt wurden. Grau dargestellt sind Eingabedaten, welche durch den Slothentwickler oder Plattformentwickler bereit gestellt werden.

2.2.3 SleepySloth

Nach der Einführung in das OSEK-BCC1-kompatible Sloth, soll nun eine Erweiterung dessen – *SleepySloth* – vorgestellt werden, welche zusätzlich *Extended Tasks* und die zugehörigen Ereignis Systemaufrufe (`SetEvent()`/`WaitEvent()`) bereitstellt. Es handelt sich hierbei weiterhin um ein Uniprozessor System, welches den OSEK-ECC1 Standard implementiert [HLSP11]. Da *Extended Tasks* blockieren können, wird in SleepySloth damit die Lücke zwischen Applikationsfäden und Unterbrechungsroutinen vollständig geschlossen. Applikationsfäden sind Unterbrechungsroutinen und Unterbrechungsroutinen sind Applikationsfäden. Das Ziel der SleepySloth-Designer war es also *Extended Tasks* anzubieten und dabei die Geschwindigkeit originalen Sloth-Betriebssystems weitestgehend zu erhalten. Auf diese Weise fällt die zusätzliche Flexibilität des Programmierers nicht zu Lasten der Geschwindigkeit.

Wie bereits in Abschnitt 2.2 erwähnt, ist der Kontrollfluss von Tasks in Sloth immer strikt gestapelt. Dieses Ausführungsmodell entspricht genau dem eines mehrschichtigen Unterbrechungskontrollers: Gerade laufenden Unterbrechungsroutinen können zwar durch höherpriore Unterbrechungsroutinen unterbrochen werden, jedoch werden diese stets in einem strikten Last-In-First-Out – also gestapelt – abgearbeitet. Es existiert somit kein Konzept im Unterbrechungskontroller, um die Ausführung von

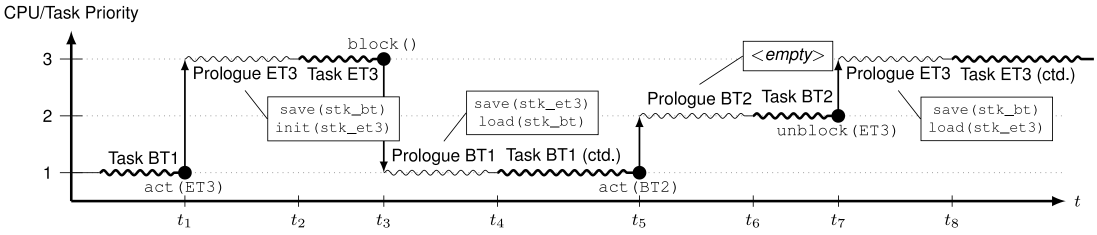


Abbildung 2.4 – Aus [HLSP11]. Gezeigt ist ein Kontrollfluss zweier *Basic Tasks* BT1 und BT2 mit Priorität 1 und 2 sowie einem *Extended Task* ET3 mit Priorität 3. BT1 und BT2 teilen sich einen den Stapel `stk_bt`, wohingegen ET3 einen eigenen Stapel `stk_et3` besitzt.

einer Unterbrechungsroutine an einer Stelle zu pausieren und später fortzusetzen.

Zur Implementation von SleepySloth muss also eine Lösung zur Deaktivierung blockierter Tasks bzw. Unterbrechungsroutinen und zur Reaktivierung nicht mehr blockierter Tasks gefunden werden. Zusätzlich besteht noch das Problem, dass durch die asynchrone Natur von Unterbrechungen, nicht vorher bestimmt werden kann, an welcher Stelle an gerade laufender Task verdrängt wird. Die Verdrängung ist transparent für den verdrängten Task, somit ist es die Aufgabe des Betriebssystems oder des verdrängenden Tasks eventuellen Kontext bzw. Stapel zu sichern. *Extended Tasks* benötigen aufgrund der Möglichkeit zur Blockierung und späteren Fortsetzung einen kompletten Kontext inklusive Stapel, wohingegen *Basic Tasks* sich Teile ihres Kontextes, z.B. den Stapel, teilen können. Um die Deaktivierung/Reaktivierung von Unterbrechungen und die Sicherung/Wiederherstellung des Ausführungskontextes zu realisieren, führt SleepySloth Task-Prologe ein.

Diese werden immer vor der eigentlichen Taskfunktion ausgeführt. Während des Prologs wird entschieden, ob und wie viel Kontext zu sichern ist, ob und wie viel Kontext wiederhergestellt werden muss und ob nicht mehr blockierte Tasks fortgesetzt werden können. So muss zum Beispiel beim Wechsel zwischen mehreren *Basic Tasks* kein Wechsel des Stapels erfolgen, beim Wechsel zu einem *Extended Task* wiederum schon (siehe Abbildung 2.4). Durch eine statische Kontrollfluss-Analyse ist es dem Sloth-Generator möglich dies bereits zur Übersetzungszeit zu erkennen, wodurch der Task-Prolog speziell auf den Anwendungsfalls zugeschnitten werden kann, d.h. im Fall von BT2 in Abbildung 2.4 vollkommen leer ist.

Der vollständige Ablauf eines Task-Prologs ist in Abbildung 2.5 gezeigt und soll aufgrund seiner Ähnlichkeit zu der Implementation in InterSloth kurz erläutert werden. Zunächst wird der Kontext des gerade unterbrochenen Tasks gesichert (① in Abbildung 2.5). Anschließend wird überprüft, ob der gerade unterbrochene Task terminiert (②) ist, falls nein, wird er erneut ausgelöst, damit seine Ausführung nach Behandlung dieses Tasks fortgesetzt wird ((2a)). Daraufhin wird der aktuell laufende Task aktualisiert (③) und überprüft, ob dieser zuvor verdrängt wurde oder ob er zum ersten Mal startet (④). Falls er zum ersten Mal startet, wird sein Kontext initialisiert ((5b)) und anschließend die Taskfunktion ausgeführt. Lief der Task bereits vorher so, wird der gesicherte Kontext wiederhergestellt ((6a)) und zur gesicherten Stelle in der Taskfunktion gesprungen ((7a)).

Die Implementierung der `WaitEvent()` Funktion ist nun trivial. Es wird zunächst geprüft ob das Ereignis, auf welches gewartet werden soll, bereits vorhanden ist. Ist dies der Fall, so wird die Ausführung des Tasks fortgesetzt. Falls nicht, wird gespeichert auf welches Ereignis gewartet werden soll, wird die zum Task gehörende Unterbrechungsquelle deaktiviert und anschließend die CPU an einen anderen Task abgegeben. Dies geschieht durch Runtersetzen der CPU-Priorität auf null und dem Reaktivieren von Unterbrechungen. Hierbei muss ähnlich zur `ActivateTask()` Funktion im Sloth-System eine gewisse Zeit gewartet werden, sodass sofort beim Reaktivieren der Unterbrechungen eine neue Unterbrechung zugestellt werden kann. Wird der blockierte Task nun von einem

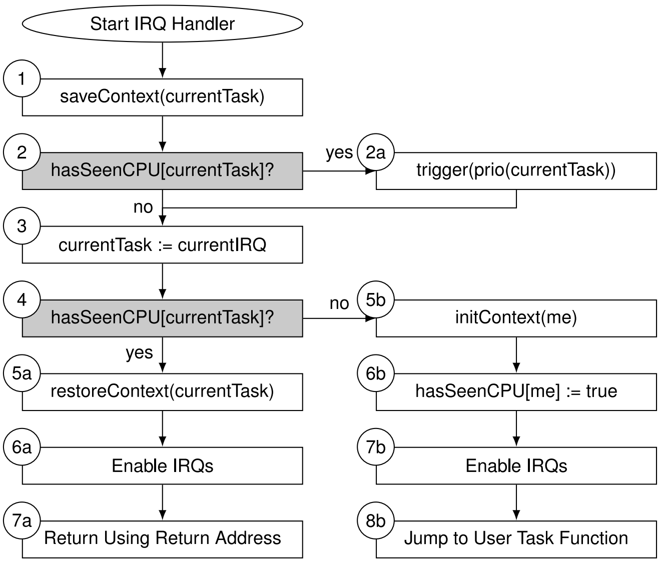


Abbildung 2.5 – Aus [HLSP11]. Kontrollflussgraph eines Prologs in SleepySloth

anderen Task verdrängt, wird die zugehörige Unterbrechungsquelle ausgelöst (siehe Schritt 2a in Abbildung 2.5). Diese Auslösung wird jedoch nicht vom Unterbrechungskontroller berücksichtigt, da die Unterbrechungsquelle in `WaitEvent()` deaktiviert wurde; der Task ist blockiert.

Analog geschieht die Implementation der `SetEvent()` Funktion. Hier wird zunächst das Ereignis als vorhanden markiert und anschließend geprüft, ob gerade Tasks auf dieses Ereignis warten. Ist dies der Fall werden die zugehörigen Unterbrechungsquellen wieder aktiviert, wodurch sie wieder in den Priorisierungsprozess des Unterbrechungskontrollers mit einbezogen werden. Anschließend wird ein gewisse Zeit gewartet – bis der Unterbrechungskontroller seine Priorisierung abgeschlossen hat –, um dann die Unterbrechungen wieder zu reaktivieren. Ab diesem Zeitpunkt ist der zuvor blockierte Task unblockiert und kann jederzeit vom Unterbrechungskontroller ausgeführt werden.

OSEK-Ressourcen bedürfen besonderer Behandlung in SleepySloth. Ein Task, welcher eine Ressource in Besitz genommen hat und somit seine Priorität auf die *Ceiling Priority* (siehe Gleichung (2.1)) angehoben hat, wird durch einen *Extended Task* verdrängt. Wird der Task später fortgesetzt, so muss er mit der Priorität der Ressource fortgesetzt werden, um das Priority Ceiling Protocol (PCP) weiter zu gewährleisten. Die Priorität einer Ressource sind jedoch im klassischen PCP als das Maximum aller Prioritäten von Tasks definiert, welche die Ressource in Besitz nehmen können. Es ist dem Betriebssystem also bei der Fortsetzung des verdrängten Task unmöglich, zwischen einem Task welcher die geteilte Ressource in Besitz genommen hat und dem höchstprioren Task der Ressourcengruppe zu unterscheiden. Aus diesem Grund wird Ressourcen bei SleepySloth ein eigene Prioritätsebene (und damit auch eine eigene Unterbrechungsquelle) zugeordnet, welche im Gegensatz zur mit Gleichung (2.1) errechneten Priorität um eine Ebene erhöht ist. Wird nun die zur Ressource gehörende Unterbrechungsquelle ausgelöst, so wird eine spezielle Ressourcenunterbrechungsroutine ausgeführt, welche den zuvor verdrängten Task wieder fortsetzt.

Durch die kontrollflusssensitive Prologgenerierung ergibt sich in einer Geschwindigkeitsevaluation keinerlei Nachteil gegenüber der originalen Sloth-Implementation sofern nur *Basic Tasks* verwendet werden. Unter Verwendung von *Extended Tasks* ergibt eine Geschwindigkeitsverbesserung zwischen 1.3-fach und 3-fach gegenüber einer kommerziellen OSEK-Implementierung. Insgesamt wurde also das gesetzte Ziel erreicht: *SleepySloth* erweitert das originale Sloth-System um *Extended Tasks* ohne dabei die Geschwindigkeitsvorteile zu verlieren.

2.2.4 MultiSloth

Sowohl *SleepySloth* als auch das originale *Sloth* sind reine Uniprozessor-Betriebssysteme. Der Trend zu Multiprozessorsystemen, welcher in Heimcomputern bereits seit mehr als einem Jahrzehnt auftritt, erreicht nun aufgrund gestiegener Leistungsanforderung und Energieeffizienzbedenken ebenfalls eingebette Systeme. Er stellt neue Anforderungen an Echtzeitbetriebssysteme, insbesondere in Bezug auf Synchronisierung und Verteilung von Ressourcen. So musste auch der weitverbreitete OSEK-Standard (siehe Abschnitt 2.1) an die geänderten Bedingungen angepasst werden. Bereits vor dem Trend hin zu Multiprozessorsystemen entstand 2003 die von führenden Automobilherstellern gegründete AUTOSAR Entwicklungspartnerschaft. Ziel war die Erweiterung und Verbesserung des weitverbreiteten OSEK-Standards (siehe Abschnitt 2.1). Der durch diese entwickelte AUTOSAR Betriebssystem Standard ist somit rückwartskompatibel zum OSEK-Standard und unterstützt seit 2014 ebenfalls Multiprozessorsysteme [AUT13]. AUTOSAR verfolgt einen partitionierten Multiprozessoransatz. Hierbei werden Task, ISRs und Alarme (im Folgenden nur als Tasks bezeichnet) statisch zur Übersetzungszeit an einen Prozessorkern gebunden und zur Laufzeit nur auf diesem ausgeführt. Die Einplanungsstrategie eines einzelnen Prozessors entspricht dabei einer der bereits aus OSEK bekannten Varianten, wobei jedoch häufig alle Prozessorkerne dieselbe Strategie ausführen.

Die statische Bindung erleichtert die Implementierung und Verifizierbarkeit des Betriebssystems auf Kosten von (potentiell) ungenutzter Kapazität. Sind Tasks suboptimal partitioniert, so kann es passieren, dass ein Prozessorkern dauerhaft ausgelastet ist, während ein anderer Kern dauerhaft z.B. auf externe Ereignisse wartet und somit ungenutzt bleibt. Die optimale Partitionierung eines Multiprozessorsystems ist zusätzlich ein NP-Schweres Problem und kann daher in vielen Fällen nur approximativ, und damit suboptimal, gelöst werden [Kop11, S. 249].

Im Gegensatz dazu funktionieren globale Multiprozessorbetriebssysteme. Hier kann ein Task jederzeit von einem Prozessorkern verdrängt werden und auf einem anderen Prozessorkern – sofort oder später – fortgesetzt werden. Es tritt somit keine ungleichmäßige Belastung auf und die durchschnittliche Antwortzeit des Betriebssystems wird verbessert. Durch Teilung von z.B. Task Kontextstrukturen über mehrere Prozessorkerne hinweg, wird jedoch ein zusätzlicher Synchronisierungaufwand nötig, wodurch Aktivierungskosten von Tasks steigen. Des Weiteren können weitere Geschwindigkeitsverluste durch Multiprozessor-Cache-Effekte auftreten.

MultiSloth ist die Multiprozessorvariante des Sloth-Systems und verfolgt einen an AUTOSAR orientierten Ansatz. Tasks, ISRs bzw. Alarme werden statisch an Prozessorkerne gebunden, wobei sich jeder Prozessorkern zur Laufzeit ähnlich zu einem Uniprozessor Sloth verhält. Unterschiede ergeben sich hauptsächlich bei einer Interprozessor-Aktivierung von Tasks, sowie der Synchronisierung von Ressourcenzugriffen über Prozessorkerne hinweg. Zusätzlich können sich Tasks, welche statisch verschiedenen Prozessorkernen zugewiesen sind, Ereignisse teilen. Hierdurch muss ein besonderes Augenmerk auf die Synchronisierung des Systemzustand während der Ereignis Systemaufrufe (`WaitEvent(), SetEvent()`) gelegt werden, um *Lost-Wakeup*-Probleme zu vermeiden.

Die *Lost-Wakeup*-Problematik tritt auf, wenn bei zwei Tasks gleichzeitig einer der Tasks aufgrund eines fehlenden Ereignisses blockiert, während der andere Task gleichzeitig jenes Ereignis auslöst. (siehe Listing 2.1). Tritt eine Unterbrechung in Zeile 6 von Listing 2.1 auf so blockiert der Task obwohl er dies nicht tun dürfte, da das Ereignis nun gesetzt ist.

Abbildung 2.6 zeigt die Struktur des MultiSloth-Betriebssystems [Mül+14]. Als Referenzplattform wurde sich für die Infineon AURIX Familien von Multiprozessor-Mikrocontrollern entschieden. Diese bieten maximal drei Prozessorkerne mit einer geteilten Speicherarchitektur an. Unterbrechungs-

```
WaitEvent(event){
    if(eventSet(event)){
      return;
    }
    else{
        // Unterbrechung welche SetEvent(event) aufruft.
        block();
    }
}
```

Listing 2.1 – *Lost-Wakeup* Beispiel

quellen werden Service-Request Nodes (SRNs) genannt und können sowohl mit externer Hardware verbunden werden als auch durch das Betriebssystem und das Timersubsystem ausgelöst werden. Ihnen kann statisch eine 8-Bit breite Priorität sowie ein Prozessorkern zugewiesen werden, welcher die Unterbrechung behandeln soll. Wird eine SRN ausgelöst, so wird die Unterbrechungsanfrage zunächst an die sogenannte Interrupt Control Unit (ICU) weitergeleitet, welche einmal pro Prozessorkern existiert. Die ICU prüft dann, ob die Priorität der SRN höher ist als die aktuelle Priorität des Prozessorkerns und unterbricht diesen gegebenenfalls. Jeder Prozessor besitzt, gemäß des partitionierten Einplanungsansatzes, eine eigene Unterbrechungstabelle und führt, nachdem er vom *ICU* unterbrochen wurde, die korrekte Unterbrechungsroutine, d.h. entweder einen Task, ISR oder Alarm aus.

Ressourcen bedürfen ebenfalls einer besonderen Behandlung in *MultiSloth*. Man unterscheidet zwischen *lokalen* Ressourcen, welche nur von Tasks innerhalb einer Partition, d.h. Prozessorkern,

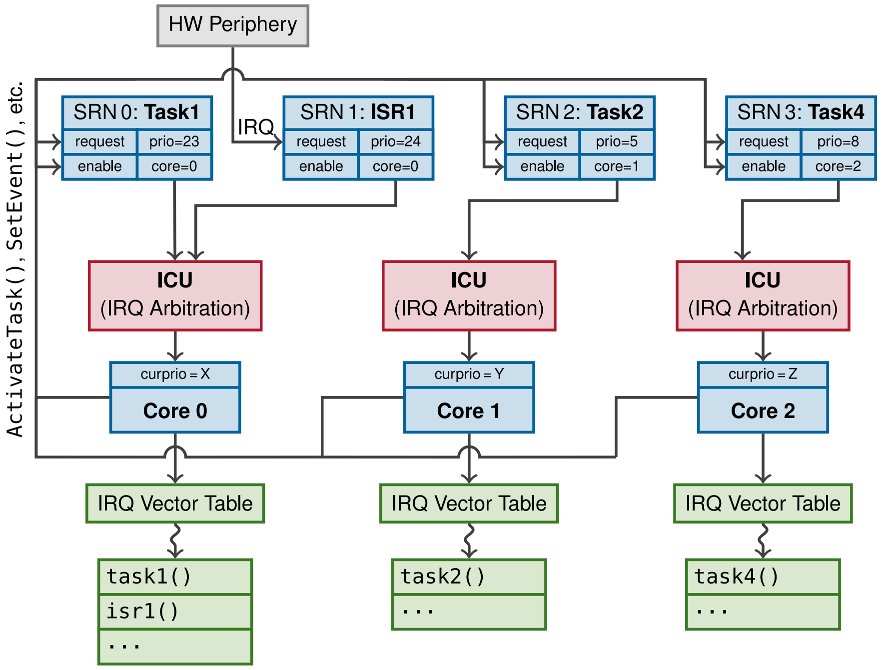


Abbildung 2.6 – Struktur des MultiSloth-Betriebssystems [Mül+14]

in Besitz genommen werden können, und *globalen* Ressourcen, welche von Tasks verschiedener Partitionen in Besitz genommen werden können. Zur Verteilung *lokaler* Ressourcen kann weiterhin das aus OSEK bekannte Priority Ceiling Protocol (PCP) (siehe Abschnitt 2.1) genutzt werden, während *globale* Ressourcen besonderer Synchronisierung bedürfen. Falls ein Task auf eine Ressource zugreifen möchte, welche bereits durch einen Task in einer anderen Partition belegt ist, so darf Letzterer nicht verdrängt werden, bevor er die Ressource wieder abgegeben hat. Andernfalls blockiert Ersterer auf unbestimmte Zeit. Der in MultiSloth implementierte AUTOSAR Standard schlägt hierfür ungeordnete Spinlocks vor, welche aber im Gegensatz zu einem wichtigen Slothmerkmal stehen: Verhinderung von Prioritätsinversionen.
Verdrängt ein mittelpriorer Task einen niederprioren Task, welcher ein globales Spinlock hält, kann hierdurch ein höherpriorer Task blockiert werden. Die Auswahl an Multiprozessorresourcenprotokollen ist vielfältig (z.B. [Blo+07] oder [Bra11]), jedoch ist *Sloth* durch die Nutzung des Unterbrechungssystems auf prioritätenbasierte Protokolle beschränkt. Aus diesem Grund entschieden sich die *MultiSloth* Entwickler dazu eine Weiterentwicklung des Priority Ceiling Protocol, das sogenannte Multicore Priority Ceiling Protocol (MPCP) [Raj90] zu implementieren, welches an dieser Stelle der Vollständigkeit halber kurz erwähnt jedoch nicht im Detail erläutert werden soll.
Die Geschwindigkeit des *MultiSloth-Systems* ist vergleichbar zu der Geschwindigkeit von *SleepySloth-Systems*. So braucht eine Taskaktivierung mit *cross-core roundtrip*, d.h. anschließender erneuter Taskaktivierung auf dem initialen Kern, in einem System aus *Basic* und *Extended Tasks* im Durchschnitt 171 Takte, was ungefähr zwei Taskaktivierung in *SleepySloth* entspricht. Insgesamt sind nur Ereignis Systemaufrufe (`WaitEvent(), SetEvent()`), aufgrund des zusätzlichen Synchronisierungsaufwands, langsamer im Vergleich zu *SleepySloth*. `ActivateTask()`, `ChainTask()` und `TerminateTask()` erzielen vergleichbare Geschwindigkeiten.

2.2.5 Zusammenfassung

Zusammenfassend zeigt die Sloth-Betriebssystemfamilie von Echtzeitbetriebssystemen also am Beispiel des Einplaners ein zentrales Konzept: Durch eine stärkere Kopplung von Betriebssystem und Hardware kann nicht nur die Betriebssystemgröße reduziert werden, sondern auch ein signifikanter Geschwindigkeitsgewinn erzielt werden, ohne dabei auf Funktionalität oder Komfort in der Entwicklung der Anwendung zu verzichten. Nachdem die Sloth-Betriebssystemfamilie eingeführt wurde, soll nun die Entwicklungsplattform RISC-V und der global-prioritätsgetreue Unterbrechungskontroller MIRQ-V eingeführt werden, der eine zentrale Vorraussetzung für die Implementierung von InterSloth darstellt.

2.3 RISC-V

RISC-V ist ein offener Befehlssatz, welche auf bekannten Reduced-Instruction Set Computer (RISC) Prinzipien basiert. Er wird seit 2010 an der Universität von Kalifornien, Berkeley entwickelt und ist unter der offenen BSD-Lizenz veröffentlicht [Fou18a]. Im Gegensatz zu anderen Open-Source-Lizenzen wie z.B der GNU General Public License (GPL), erlaubt die BSD-Lizenz eine Veröffentlichung von abgeleiteten Arbeiten unter anderen Lizenzbedingungen. Auf diese Weise ist es nicht nötig, dass der Quellcode von Prozessordesigns, welche auf dem RISC-V-Befehlssatz basieren, veröffentlicht werden. Firmen wie *ARM* oder *IBM* erlauben, nach Zahlung von Lizenzgebühren, ebenfalls die Nutzung und Erweiterung ihrer Befehlssatzarchitekturen. Kosten im Bereich von bis zu zehn Millionen USD und einer Verhandlungsphase von bis zu zwei Jahren machen kommerzielle Architekturen jedoch ungeeignet für den Einsatz in Lehre und Forschung [AP14]. Bisherige offenen Architekturen wie

DLX [Pö] oder OpenRISC [oA17] sind entweder stark veraltet oder besitzen andere architekturielle Nachteile, welche eine kommerzielle Implementierung erschweren.
Der RISC-V-Gründer Krste Asanović ist jedoch der Meinung, dass eine kommerzielle Verwendung einer Architektur erst das nötige Interesse an dieser weckt um eine gemeinsame Weiterentwicklung durch die Open-Source-Gemeinschaft zu ermöglichen. So entwickelte er 2010 zusammen mit seinen Doktoranden den ersten Entwurf des RISC-V-Befehlssatzes, welcher genauso für einen 16-Bit Mikrocontroller wie ein Warehouse-Rack mit mehreren hundert Kernen geeignet sein sollte. Unterstützt wurde er von RISC-Begründer David Patterson, welcher ebenfalls an der Universität von Kalifornien beschäftigt ist. Ursprünglich sollte das Projekt nach Asanović's Aussage nur ein „kurzes, dreimonatiges Projekt über die Sommerferien“ werden [Asa16]. Nach positiver Rezeption wurde 2010 schließlich die RISC-V-Stiftung, unterstützt von Chipherstellern wie AMD oder NVIDIA, gegründet, welche nun die Weiterentwicklung des RISC-V-Befehlssatz finanziert und verwaltet. Viele Mitglieder der RISC-V-Stiftung verwenden mittlerweile intern RISC-V-basierte Prozessoren [Dig17]. Dies ist auch der Grund, warum RISC-V für so viele Anwendungszwecke portiert wurde: Eine breite Anwendungsbasis führt insgesamt zu mehr Beiträgen zur Architektur.
Neben wenig Interesse stellt häufig auch mangelnde Softwareunterstützung ein großes Problem für neuen Architekturen da. Durch die breite Industrieunterstützung gibt es jedoch bereits heute Portierungen des Linux Kernels, samt kompletter Toolchain (*binutils*) und Kompiler (*gcc* und *llvm*). Zurzeit existieren mehrere Implementationen des RISC-V-Befehlssatzes [Fou18b]:

- 64-Bit *Rocket Core* geschrieben in der Hardwarebeschreibungssprache Chisel
- fünf verschiedene 32-Bit *Sodor* Prozessoren, Universität von Kalifornien, Berkeley
- 32-Bit PULPino, ETH Zurich

Des Weiteren existieren kommerziell verfügbare Chips von z.B. Andes Technology Corporation, sowie SiFive Incorporated, sowie verschiedene Simulationsframeworks wie z.B. QEMU oder gem5 bereits für RISC-V.
Im Folgenden soll nun zuerst eine Übersicht über die RISC-V-Architektur gegeben werden, um anschließend genauer auf das Unterbrechungssubsystem einzugehen.

2.3.1 Architekturübersicht

Die Spezifikation der RISC-V-Architektur gliedert sich in zwei Teile [WAI17b]:

User-Level Befehlssatz definiert Arithmetik, Instruktionsenkodierung, Register und Speicherarchitektur (siehe [WAI17a]).

Privileged Befehlssatz definiert betriebssystemrelevante Funktionalität, d.h. Unterbrechungssystem, Speicherschutz und Privilegienlevel (siehe [WAI17b]).

Diese Aufteilung wurde durch die RISC-V-Stiftung gewählt um, Entwicklungen im Bereich der Betriebssysteme unabhängig von Übersetzer- und Anwendungsentwicklern zu machen. RISC-V ist eine little-endian Architektur, welche in der Basisvariante eine Wortgröße von 32-Bit benutzt und standardmäßig 32 Register enthält (siehe Tabelle 2.2). Im Gegensatz zu vielen bekannten Architekturen besitzt RISC-V kein Fehlercode- bzw. Übertragsregister, um z.B. eine Division durch 0 oder einen Überlauf zu detektieren. Es existieren neben verschiedensten Gleitkommaerweiterungen ebenfalls 64-Bit und 128-Bit Erweiterungen der Basisarchitektur.
Der RISC-V-Befehlssatz ist eine von Grund auf erweiterbare Architektur, wobei der unterstützte Funktionsumfang durch eine Kombination aus Zahlen und Großbuchstaben encodiert wird. Der

Erweiterung	Beschreibung
I	Basis Befehlssatz, Integer Arithmetik und Speicherzugriff
M	Integer Multiplikation/Division
A	Atomare Operationen
F, D, Q	Single-, Double-, Quadprecision Gleitkommazahlen
C	Komprimierte Instruktionen
G	Zusammenfassung von I, M, A, F, D

Tabelle 2.1 – RISC-V-Befehlssatz Erweiterungen

Name beginnt immer mit dem Präfix *RV*, woraufhin die Basiswortgröße in Bit folgt. Anschließend werden – je nach unterstützten Funktionsumfang – Großbuchstaben angehängt. Eine Übersicht einiger valider Großbuchstaben bietet Tabelle 2.1 (gezeigt sind nur „frozen“ Erweiterungen, d.h. nur Erweiterungen dessen Spezifikation bereits finalisiert wurde). Die Basisarchitektur, welche Instruktionen zur Integerarithmetik und Speicheroperation enthält, wird hierbei durch den Präfix *I* bezeichnet. Weiterhin werden die in kommerziellen Chips häufig implementierten Erweiterungen *I*, *M*, *A*, *F* und *D* zwecks der Lesbarkeit in der Erweiterung *G* zusammengefasst. Das Speichermodell von RISC-V ist das einer *load-store* Architektur, d.h. die einzigen Befehle, welche auf den Hauptspeicher zugreifen können, sind `load` und `store`, zum Lesen und Schreiben von Speicher respektive. Sie existieren – je nach Basislänge des Befehlssatzes – in 1, 2, 4, 8 oder 16 Byte Varianten. Sämtliche Arithmetik und Logik findet somit nur mit Registern oder Konstanten statt. Im Gegensatz zu bekannten kommerziellen Architekturen, wie IA-32 (Intel 32-Bit Architektur), werden Funktionsargumente, Rückgabewerte sowie Rahmenzeiger und Rücksprungadressen in Registern übergeben. Diese Maßnahmen sollen Speicherzugriffe minimieren und so zu einem Geschwindigkeitsvorteil führen. Das Speicherkonsistenzmodell in Bezug auf mehrere parallele Ausführungskontexte ist relaxiert. Jeder Ausführungskontext observiert Speicherzugriffe innerhalb seines eigenen Kontextes, d.h. Programmausführung, sequentiell. In welcher Ordnung die eigenen Speicheroperationen anderen Kontexten sichtbar gemacht werden, ist jedoch nicht garantiert. Zur Synchronisierung existieren sogenannte `fence` Instruktionen, welche nach ihrer Ausführung garantieren, dass alle Instruktion bzw. Speicheroperationen vor ihnen erfolgreich anderen Kontexten sichtbar gemacht wurden. Es

Registername	ABI Name	Funktion
`x0`	`zero`	Konstant 0
`x1`	`ra`	Rücksprungadresse
`x2`	`sp`	Stapelzeiger
`x3`	`gp`	Globaler Zeiger
`x4`	`tp`	Threadzeiger
`x5–7`	`t0–2`	Temporäre Register
`x8`	`s0/fp`	Gespeichertes Register/Framepointer
`x9`	`s1`	Gespeicherte Register
`x10–11`	`a0–1`	Funktionsargumente/Rückgabewerte
`x12–17`	`a2–7`	Funktionsargumente
`x18–27`	`s2–11`	Gespeicherte Register
`x28–31`	`t3–6`	Temporäre Register

Tabelle 2.2 – Register des RISC-V-Befehlssatzes

gibt verschiedene `fence` Instruktionen, je nachdem für welche Arten von Speicherzugriffen, d.h. reine Speicherzugriffe oder speicherabgebildete Ein/Ausgabe, synchronisiert werden sollen.
RISC-V unterscheidet zwischen Prozessoren und *Hardware-Threads* (Harts). Ein Prozessor führt durch Multithreading potentiell mehrere Harts parallel aus, wodurch sich Instruktionen, welche den Ausführungskontext verändern, stets auf einen Hart beziehen und nicht zwangsweise auf einen Prozessor.
Jeder Hart enthält einen eigenen Satz Register, einen Programmzeiger und einen eigenen Satz von Kontroll- und Statusregister (CSR). CSRs werden in dem *Priviliged Befehlssatz* definiert und werden durch spezielle Instruktionen (`csrr, csrw`) gelesen und geschrieben. Neben den in Tabelle 2.3 erwähnten CSRs existieren noch viele weitere CSRs, auf welche z.B. in Abschnitt 2.3.2 noch weiter eingegangen wird.
Ein Hart läuft zu jedem Zeitpunkt auf einem der vier verfügbaren Privilegienebenen (siehe Tabelle 2.4). Der Maschinenmodus stellt die höchste Privilegienstufe dar und hat unbeschränkten Zugriff auf die Hardware bzw. die CSRs. Er muss als einzige Privilegienebene zwangsweise vorhanden sein, wohingegen es Chipherstellern, z.B. für eingebette Systeme freisteht, keinen User bzw. Supervisor Modus anzubieten. RISC-V unterstützt außerdem die Delegierung von Ausnahmen (sogenannten *Traps*) über verschiedene Privilegienebenen hinweg. Somit ist es möglich, Unterbrechungen bzw. Ausnahmen – je nach Typ – in verschiedenen Ebenen zu behandeln.

2.3.2 Unterbrechungssubsystem

Das Unterbrechungssubsystem in RISC-V unterscheidet zwischen zwei Arten von Unterbrechungen [WAI17b]:

- *Lokalen Unterbrechungen*, d.h. Unterbrechungen, welche direkt an einen zuvor bestimmten Hart zugestellt werden.
- *Globalen Unterbrechungen*, d.h. Unterbrechungen, für welche dynamisch ein Zielhart bestimmt wird.

Lokale Unterbrechungen, zu denen auch Timer- und Softwareunterbrechungen gehören, werden durch Core-Local Interruptor (CLINT) bearbeitet und ausgelöst. *Globale Unterbrechungen* werden hingegen von dem Platform-Level Interrupt Controller (PLIC) bearbeitet und weitergeleitet (siehe

CSR Name	Zugriffsart	Funktion
`marchid`	RO	Architektur ID
`mvendorid`	RO	Hersteller ID
`mimpid`	RO	Eindeutige ID der Hardware Revision
`mhartid`	RO	Eindeutige ID des aktuell laufenden Harts
`misa`	RW	Implementierte Befehlssatzerweiterungen
`mstatus`	RW	Zustand des Harts, z.B. Privilegienebene, Unterbrechungszustand
`mie`	RW	Deaktivierung einzelner Unterbrechungsarten
`mip`	RW	Auslösestatus einzelner Unterbrechungsarten
`mtvec`	RW	Unterbrechungsbehandleradresse
`mcause`	RW	Grund für Unterbrechung
`mepc`	RW	*Program counter* bei Unterbrechung

Tabelle 2.3 – Ausgewählte RISC-V CSRs

Level	Modus Name	Abkürzung
0	User	U
1	Supervisor	S
2	*Reserviert*	
3	Machine	M

Tabelle 2.4 – Privilegienebenen des RISC-V-Befehlssatzes

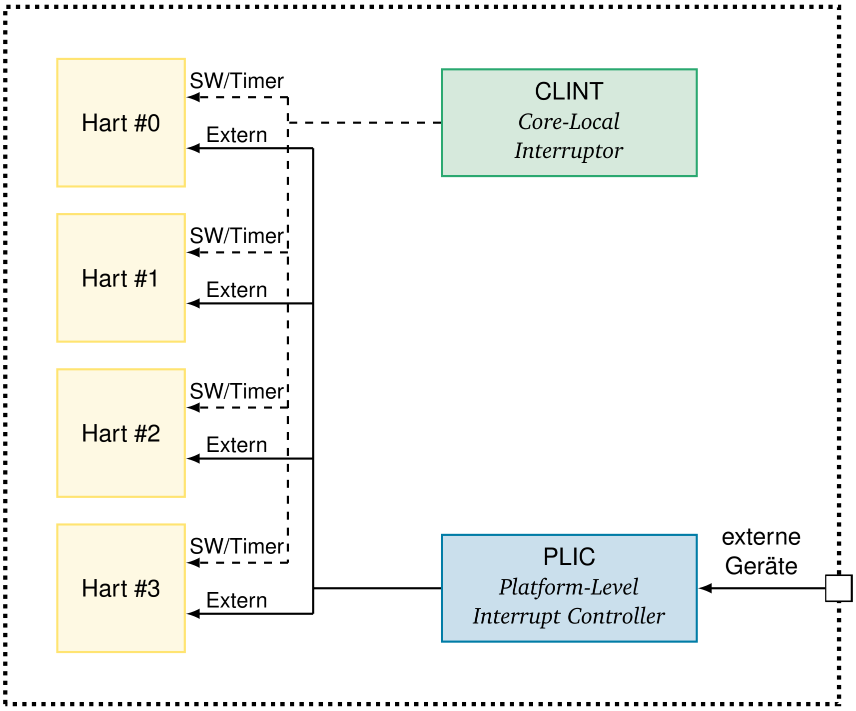


Abbildung 2.7 – Struktur des Unterbrechungssystem unter RISC-V

Abbildung 2.7). Für diese Trennung von Hart und Unterbrechungsverwaltung entschieden sich die RISC-V-Designer, um eine große Flexibilität in der Unterbrechungsbehandlung zu ermöglichen. So kann der PLIC, je nach Anwendungsfall, verschiedene Unterbrechungspriorisierung und Priorisierungsschemata unterstützen, ohne dass hierfür das Design des Prozessors angepasst werden muss. Da die Modifikation des PLICs ein essentieller Bestandteil dieser Arbeit ist, soll er nun im Detail erläutert werden.

Der PLIC unterstützt verschiedene Prioritätsebenen für Unterbrechungsquellen, sowie eine Prioritätsschwelle (*threshold*) pro Hart, welcher das Ausblenden von Unterbrechungsquellen unterhalb einer Prioritätsebene ermöglicht. Die Konfiguration des PLICs ist ähnlich zu bekannten Architekturen speicherzugeordnet (*memory-mapped*), d.h. die Priorität einer Unterbrechungsquelle, sowie ihr Aktivierungzustand und Prioritätsschwelle jedes Harts werden durch Lesen und Schreiben spezieller Speicherstellen konfiguriert. Die erste Unterbrechungsquelle (Index 0) kann nicht konfiguriert werden und entspricht in der RISC-V-Semantik keiner Unterbrechung, sondern ist vielmehr das Analogon zu „keiner Unterbrechung“ .

Jede Unterbrechungsquelle kann zusätzlich einzeln pro Hart deaktiviert und aktiviert werden, hat jedoch eine globale Prioritätsebene. Grundsätzlich unterstützt der PLIC zur Zeit einen *unicasting* Ansatz zur Unterbrechungszustellung, d.h. zunächst wird jeder Hart unterbrochen, dessen Prioritätsschwelle unterhalb der Priorität der gerade ausgelösten Unterbrechung liegt. Die tatsächliche Behandlung der Unterbrechung findet jedoch nur in einem Hart statt (siehe Abschnitt 2.3.2.2). Standardmäßig werden Unterbrechungen nur im Maschinenmodus behandelt, jedoch erlaubt RISC-V eine Delegierung der Unterbrechungen an höhere Privilegienebenen durch Konfiguration spezieller CSRs. Neben der globalen Deaktivierung von Unterbrechung durch Löschen des `MIE` Bits im CSR `mstatus`, kann jede Unterbrechungsart (Timer, Software, Extern oder verschiedene Ausnahmearten) durch Löschen des entsprechenden Bits im CSR `mie` deaktiviert werden.

2.3.2.1 Behandlung von Unterbrechungen

Abbildung 2.8 zeigt den Ablauf der Unterbrechungsbehandlung unter RISC-V. Wird eine Unterbrechung an einen Hart zugestellt, so wird zunächst der Wert des `mstatus.MIE` Bits, welches Unterbrechungen global aktiviert, in `mstatus.MPIE` gesichert (① in Abbildung 2.8). Anschließend wird es auf null gesetzt um Unterbrechungen auf dem Hart zu deaktivieren (②). Die Privilegienebene, welche der Hart innehatte als er unterbrochen wurde, wird ebenfalls im `mstatus` Register abgelegt (③), um dann den Ausnahmecode in `mcause` zu speichern (④). Schließlich wird der Program Counter (PC) in `mepc` gesichert und schließlich, je nachdem ob Vektorunterbrechungen aktiviert sind (⑥), auf die Adresse der Unterbrechungsbehandlungsroutine, welche in `mtvec` abgelegt ist, gesetzt (⑤, (7a), (7b)). RISC-V unterstützt sowohl einen Vektorunterbrechungsmodus, in dem, abhängig von

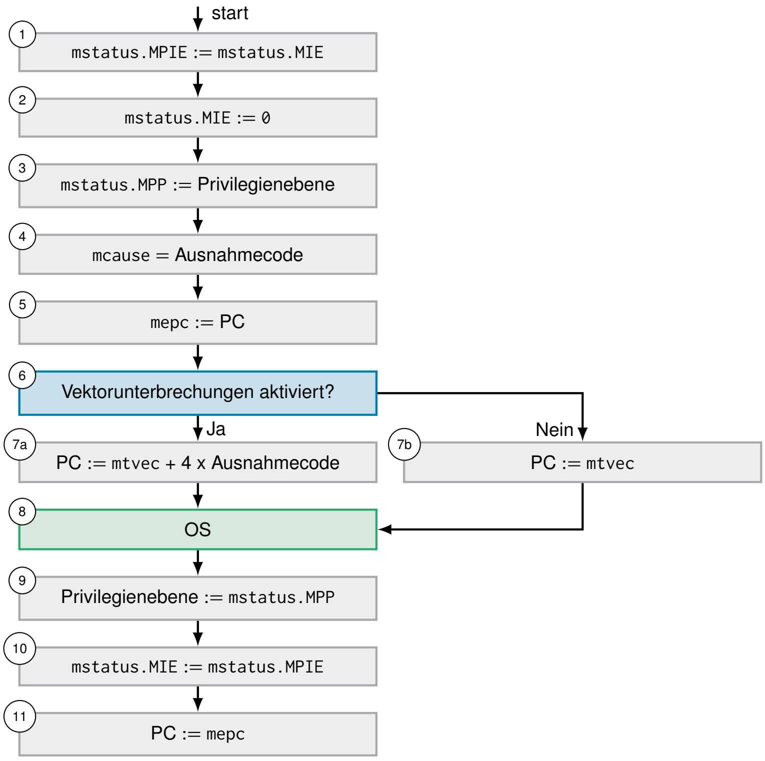


Abbildung 2.8 – Flussdiagramm zur Unterbrechungsbehandlung unter RISC-V

der Art der Unterbrechungen, eine unterschiedliche Programmadresse zur Behandlung angesprungen wird, als auch einen statischen Modus in dem, unabhängig von der Art der Unterbrechung, eine feste Programmadresse angesprungen wird. Hierbei sei angemerkt, dass verschiedene externe Unterbrechungen (d.h. durch den PLIC verwaltete) stets dieselbe Adresse anspringen, da sie alle externe Unterbrechungen sind und es dem Hart nicht möglich ist, zu diesem Zeitpunkt zwischen verschiedenen externen Geräten zu unterscheiden. Nach der Behandlung der Ausnahme durch das Betriebssystem (⑧) kann dieses durch die Ausführung der `mret` Instruktion die Kontrolle zurück an die Hardware übergeben. Diese stellt dann die zuvor gesicherte Privilegienebene (⑨) und das gesicherte `mstatus.MIE` Bit wieder her (⑩) und springt an den zuvor in `mepc` gespeicherten PC zurück(⑪). Tabelle 2.3 bietet eine Übersicht der an der Unterbrechungsbehandlung beteiligten CSRs.

Es ist ebenfalls möglich, die gerade beschriebene asynchrone Behandlung von Unterbrechungen zu deaktivieren und stattdessen zu geeigneten Zeitpunkten im CSR `mip` zu überprüfen, welche Arten von Unterbrechung gerade ausgelöst sind und diese entsprechend zu behandeln.

2.3.2.2 Unterbrechungsbehandlung im PLIC

Nachdem die grundlegende Unterbrechungsbehandlung näher gebracht wurde, soll in diesem Abschnitt auf die Behandlung von externen Unterbrechungen, d.h. der Unterbrechungen, welche im PLIC verwaltet werden, genauer eingegangen werden. Abbildung 2.10 zeigt die Kommunikationsstruktur zur Behandlung externer Unterbrechungen.

Nach dem Anspringen der Unterbrechungsbehandlungsroutine dekodiert der unterbrochene Hart zunächst die Art der Unterbrechung aus dem `mcause` CSR. Handelt es sich um eine externe Unterbrechung, so muss zunächst ermittelt werden, welche Unterbrechungsquelle die Unterbrechung ausgelöst hat. Dieser auch *claim* genannte Vorgang geschieht durch das Lesen des speicherzugeordneten `claim/complete` Registers, welches genau einmal pro Hart in dem Speicherplan des PLICs existiert. Der PLIC gibt hierbei stets die aktuell höchstpriore Unterbrechungsquelle zurück, welche als ausgelöst markiert wurde. Es kann also bei schnell aufeinander folgenden Aktivierung zweier Unterbrechungsquellen passieren, dass die zurückgegebene Unterbrechungsquelle nicht der Unterbrechungsquelle entspricht, welche die Unterbrechung des Hart ausgelöst hat.

Nach dem erfolgreichen Lesen der Unterbrechungsquelle wird diese intern als bearbeitet markiert und jedes weitere Lesen des `claim/complete` Registers gibt Unterbrechungsquelle 0, d.h. „keine Unterbrechung“ , zurück. Das Gleiche kann ebenfalls passieren, wenn durch die *unicasting* Arbeitsweise des PLICs mehrere Harts gleichzeitig darum konkurrieren die Unterbrechung zu bearbeiten. Nach der Bearbeitung der Unterbrechung durch den Hart markiert dieser die Unterbrechung als abgeschlossen durch Rückschreiben des Unterbrechungsquellenindexes in das `claim/complete` Register. Dieser Vorgang wird als *complete* der Unterbrechung bezeichnet. Erst ab diesem Punkt können wieder Unterbrechung dieser Unterbrechungsquelle ausgelöst werden. Der Zustand jeder Unterbrechungsquelle kann also endlicher Automat aufgefasst werden (siehe Abbildung 2.9), bei dem der Zustand einer Unterbrechungsquelle ständig zwischen Ausgelöst, in Bearbeitung und dem Leerlauf hin und her wechselt.

2.4 MIRQ-V

Ähnlich zu dem im RISC-V verwendeten PLIC, haben viele kommerzielle Unterbrechungskontroller zwei Einschränkungen, welche der Implementierung eines globalen Einplanungsansatzes mit Hilfe des Unterbrechungskontroller im Weg stehen [Bew16]:

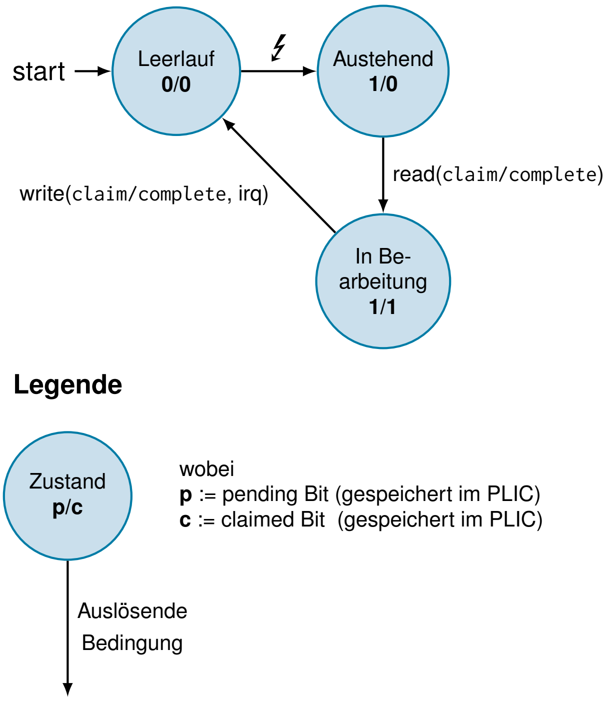


Abbildung 2.9 – Zustandsautomat von Unterbrechungsquellen

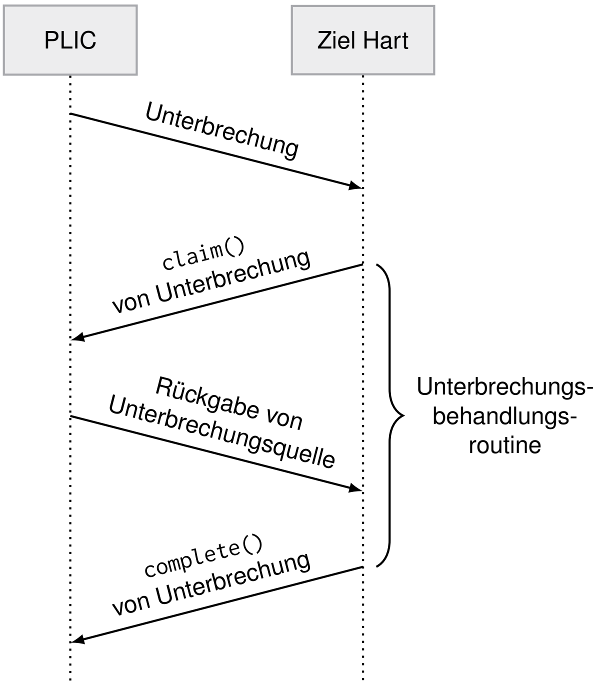


Abbildung 2.10 – Unterbrechungsablauf im PLIC, adaptiert aus [WAI17b]

- Sie geben keine Garantie, dass eine Unterbrechung an den global niedrigstprioren Prozessor zugestellt wird.
- Es gibt keine Möglichkeit eine bereits in Bearbeitung befindliche Unterbrechung zurückzugeben, um sie erneut (z.B. an einen anderen Prozessor) zuzustellen (auch bekannt unter Unterbrechungsmigration)

Um Prioritätsinversionen zu verhindern, ist es jedoch in einem globaleinplanenden System nötig, Unterbrechungen an den global niedrigstprioren Prozessor zuzustellen. Gleichfalls wird eine Unterbrechungsmigration benötigt, um die strikte Prioritätstreue im Falle einer Verdrängung zu garantieren. Aus diesem Grund setzte sich Christian Bewermeyer 2016 in seiner Bachelorarbeit zum Ziel, einen strikt global-prioritätsgetreuen Unterbrechungskontroller zu entwickeln, welcher die Möglichkeit zur Unterbrechungsmigration besitzt. Es entstand ein in der Hardwarebeschreibungssprache Chisel geschriebener Unterbrechungskontroller mit dem Namen MIRQ-V.
Neben strikter Prioritätseinhaltung und Unterbrechungsmigration, sollte er trotzdem grundlegende Funktionen wie Unterbrechungsprioritäten und dynamische Prozessorprioritäten unterstützen. Diese funktioniert ähnlich zu dem aus dem PLIC bekannten Prozessorprioritätsschwelle:
Unterbrechungen mit einer Priorität, welche kleiner ist als die Priorität des Prozessors, können diesen nicht unterbrechen.
Eine dynamische Prozessorpriorität kann jedoch in vielen Fällen zu Wettlauf-Situationen führen. Wird eine Unterbrechung an einen Prozessor zugestellt, welcher gerade seine Priorität ändert, so kann es passieren, dass nach dieser Änderung der Prozessor kein valides Ziel mehr für die Unterbrechung darstellt. MIRQ-V fängt diesen Fall ab und stellt solche inkorrekt zugestellte Unterbrechungen

automatisch erneut an ein valides Ziel zu. Grundsätzlich erlaubt MIRQ-V jedoch nur einem Prozessor gleichzeitig die Änderung seiner Priorität.
Abbildung 2.11 zeigt das Interface zwischen dem MIRQ-V Unterbrechungskontroller und einem Prozessor. Die Pfeilrichtung ist hierbei äquivalent zur Kommunikationsrichtung. Ähnlich zur Unterbrechungsbehandlung im PLIC kann sich eine Unterbrechungsquelle zu jeder Zeit in einem von drei Zuständen befinden: nicht ausgelöst, ausgelöst und in Bearbeitung. Intern prozessiert MIRQ-V in jedem Takt nur die höchstpriore, ausgelöste, aber noch nicht in Bearbeitung befindliche, Unterbrechung. Sie wird mit Hilfe eines rekursiven Algorithmus dem niedrigstprioren Prozessor zugewiesen. Daraufhin wird der Index der Unterbrechung in einen Puffer kopiert, welcher als aktuelle Unterbrechungsquelle am Prozessor anliegt, und schließlich der Prozessor unterbrochen. Der Prozessor markiert anschließend die Unterbrechung in Bearbeitung durch Anlegen eines hohen Pegels an die „ Annahme der Unterbrechung“ Leitung. Auf die gleiche Art und Weise kann eine Unterbrechung als Behandelt markiert werden oder eine erneute Zustellung der Unterbrechung angefordert werden.
Durch die Auswahl von Chisel als Entwicklungssprache konnte der MIRQ-V Unterbrechungskontroller mit Hilfe des Chisel C++-Simulators, einem taktakkuraten Register-Transfer Simulator, evaluiert werden. MIRQ-V zeigte neben einer funktionalen Korrektheit eine besonders geringe Unterbrechungslatenz von nur vier Takten, welche sich auch im Fall von vielen Prozessoren nicht erhöhte.

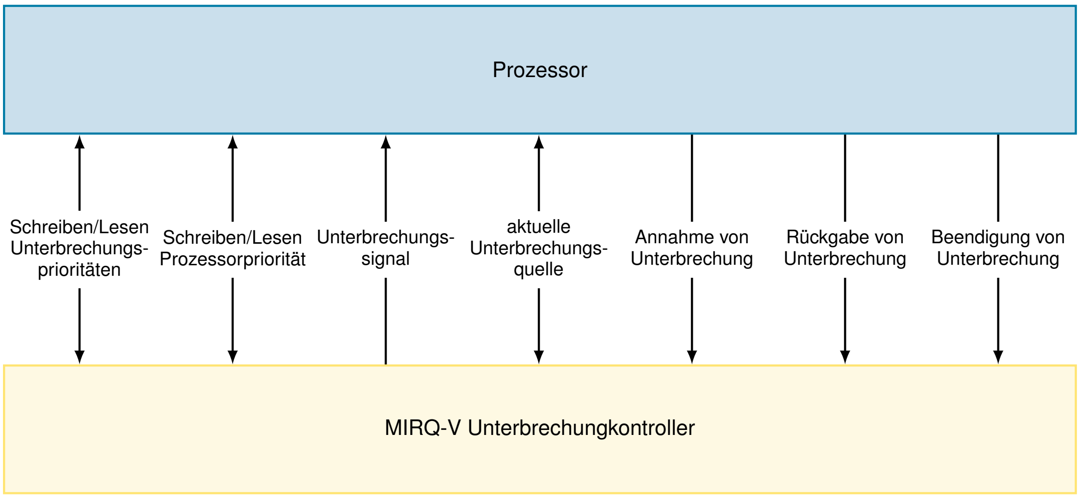


Abbildung 2.11 – Schnittstelle zwischen MIRQ-V und Prozessor. Die Pfeilrichtungen geben die Kommunikationsrichtung an.

ARCHITEKTUR 3

Nachdem im vorherigen Kapitel die Grundlagen der Sloth-Betriebssystemfamilie, der RISC-V-Architektur und schließlich des global-prioritätsgetreuen Unterbrechungskontroller MIRQ-V erklärt wurden, soll in diesem Kapitel gezeigt werden, wie mit Hilfe dieser eine Variante des Sloth-Betriebssystems – InterSloth – implementiert werden kann, welches einen globalen Einplanungsansatz verfolgt. Hierfür ist die Erweiterung des PLICs um globale Prioritätstreue sowie Unterbrechungsmigration nötig. Abschnitt 3.1 beschäftigt sich mit der Erweiterung des PLICs zur Inklusion der MIRQ-V Funktionalität. Anschließend wird in Abschnitt 3.2 schließlich, nach einem kurzen Architekturüberblick, auf die tatsächliche Implementierung der InterSloth-Variante des Sloth-Betriebssystems eingegangen.

3.1 Erweiterung des PLICs

Der RISC-V-Befehlssatz spezifiziert mehrere Unterbrechungskontroller, den CLINT zur Behandlung lokaler Unterbrechungen und den PLIC zur Behandlung globaler Unterbrechungen. Während der CLINT Unterbrechungen statisch an einen zuvor bestimmten Hart zustellt, selektiert der PLIC das Ziel der Unterbrechungsbehandlung dynamisch zur Laufzeit. Des Weiteren unterstützt der PLIC verschiedene Prioritätsebenen für Unterbrechungen, sowie eine rudimentäre Maskierung von Unterbrechungen durch eine Schwellwert-Priorität pro Hart, unter welcher der Hart nicht unterbrochen werden kann. Zur Implementierung der durch den MIRQ-V beschriebenen Funktionalität wurde daher der PLIC ausgewählt. Um den PLIC an die neue Funktionalität anzupassen, muss zunächst der in Abbildung 2.9 beschriebene endliche Automat für eine Unterbrechung erweitert werden (siehe Abbildung 3.1). Um die Darstellung übersichtlich zu halten, wurde auf alle Zustandsübergänge verzichtet, welche denselben Ein- und Ausgangszustand haben.

Es wird ein neuer Unterbrechungszustand („Zugestellt “) hinzugefügt, welcher angenommen wird, sobald eine Unterbrechung an einen Prozessor zugestellt wurde, jedoch bevor dieser tatsächlich mit der Bearbeitung der Unterbrechung begonnen hat. Dieser Zwischenzustand ist nötig, um Wettlauf-Situationen zu vermeiden, wenn eine Unterbrechung zugestellt wurde, während ein Prozessor seine Priorität ändert. Tut er dies, so muss die Unterbrechung immer dann neu zugestellt werden, sofern die neue Prozessorpriorität größer ist als zuvor, und ebenfalls größer als die Priorität der zugestellten Unterbrechung ist. In diesem Fall ist der ausgewählte Hart kein valides Ziel mehr für die bereits zugestellte Unterbrechung (① in Abbildung 3.1).

Jeder Unterbrechungszustand ist somit durch 3 Bits encodierbar (pending, delivered und claimed), welche sich jeweils darauf beziehen, ob eine Unterbrechung ausgelöst, an einen Hart zugestellt oder sich gerade in Bearbeitung befindet.

Des Weiteren werden zwei neue Zustandsübergänge hinzugefügt:

1. Es kann eine erneute Zustellung der Unterbrechung angefordert werden (② in Abbildung 3.1).

2. Es ist möglich, manuell Unterbrechungen anzufordern (③ in Abbildung 3.1).

Die Möglichkeit manuell Unterbrechungen auszulösen ist nicht in der MIRQ-V Spezifikation enthalten, erweist sich jedoch in Bezug auf das Sloth-Betriebssystem als notwendig, da dort, um ein Task zu Aktivieren, eine Unterbrechung aus der Software ausgelöst werden muss.

Das Hauptziel bei der Erweiterung des PLICs war die Erhaltung der bekannten Konfigurations- und Bedienungssemantik. Durch das ungünstige Layout des Speicherplans (*memory-map*) des PLICs ist es nicht möglich, zwei weitere hartspezifische, d.h. einmal pro Hart existierende, speicherzugeordnete Register hinzuzufügen ohne die Abwärtskompatibilität des modifizierten PLICs zu gefährden. Alle drei Statusänderungen, die mit einer Unterbrechung geschehen können, d.h. Beendingung, Neuzustellung und Auslösung, brauchen zudem Informationen darüber um welche Unterbrechung es sich handelt, bei der der Status angepasst werden soll. Es bietet sich deswegen an, alle drei Operationen über die gleiche Schnittstelle verfügbar zu machen. Aus diesem Grund wurde entschieden, sich auf eine maximale Anzahl von Unterbrechungsquellen (1024) zu einigen, um dann noch unbenutzte Bits im `claim/complete` Register als Kodierung für die durchzuführende Statusänderung zu nutzen. Um 1024 Unterbrechungsquellen abzubilden, werden 9 Bit zur Identifizierung der Unterbrechung benötigt. Es gibt insgesamt drei Operationen, somit werden zwei weitere Bits zu Encodierung der Operation benötigt. Abbildung 3.2 zeigt die Anordnung der Bits im `claim/complete` Register.

In seiner unmodifizierten Variante erwartet der PLIC lediglich einen Unterbrechungsindex beim Schreiben in das `claim/complete` Register. Bei der Encodierung der Operation wurde daher entschieden die Beendigung einer Unterbrechung mit der Binärzahl `00` zu encodieren um eine Abwärtskompatibilität des modifizierten PLICs zu garantieren. Tabelle 3.1 zeigt alle Encodierungen der Unterbrechungsoperationen.

Um die aus MIRQ-V bekannten CPU-Prioritäten zu implementieren wurde das speicherzugeordnete `threshold` Register um Funktionalität erweitert. Es handelt sich nun mehr nicht um eine Prioritätenschwelle, welche angibt, ab welcher Prioritätenebene der zugehörige Hart unterbrochen werden kann, sondern um ein Register, welches angibt auf welcher Prioritätenebene sich der Hart befindet. Aus diesem Grund wurde das Register von `threshold` zu `priority` umbenannt. Bearbeitet ein Hart eine Unterbrechung, so enthält das `priority` Register die Priorität der Unterbrechung, welche gerade Bearbeitung ist. Um den Hardwareaufwand einer potentiellen Implementierung des erweiterten PLICs gering zu halten, wurde jedoch auf eine automatische Zurücksetzung der Priorität bei Neuzustellung oder Komplettierung der Unterbrechung verzichtet. Bei einer beliebigen Stapelung von Unterbrechungen ist die Anzahl an vorherigen Prioritätsebenen nicht begrenzt und eine Implementierung in Hardware ausgeschlossen. Es liegt somit in der Verantwortung des Betriebssystems, eventuell veraltete Werte im `priority` Register zu invalidieren.

Operation	binäre Encodierung	symbolischer Name
Beendigung	`00`	`OP_COMPLETE`
Neuzustellung	`01`	`OP_OUTOFSERVICE`
Auslösen	`10`	`OP_TRIGGER`

Tabelle 3.1 – Encodierung von Unterbrechungsoperationen

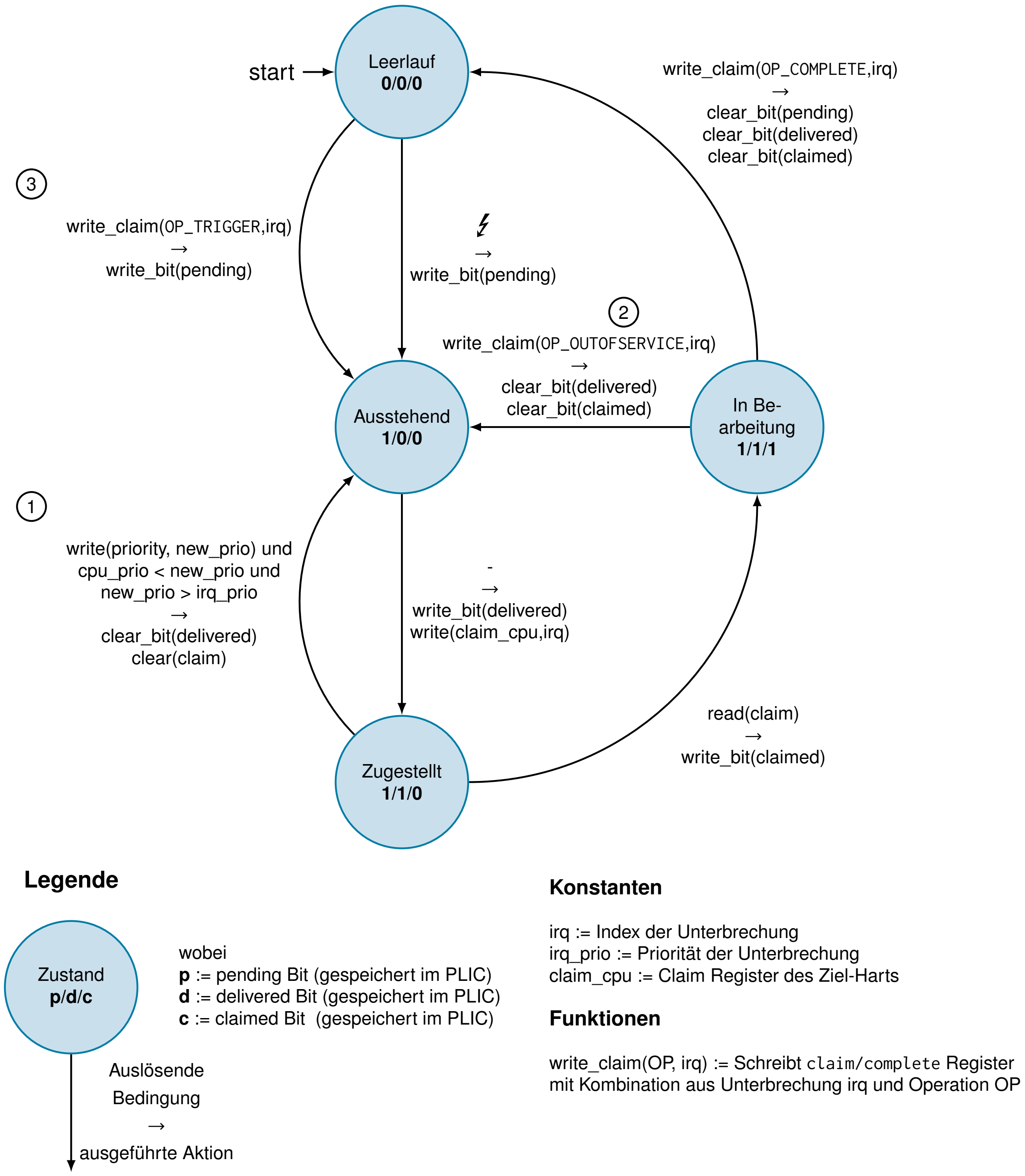


Abbildung 3.1 – Erweitertes Zustandsdiagramm **einer** Unterbrechung im PLICs

XLEN 12	11 9	8 0
Reserviert	`Op`	`Unterbrechung`
(XLEN – 11) *Bit*	2 *Bit*	9 *Bit*

Abbildung 3.2 – Layout von modifizierten `claim/complete` Register

3.2 InterSloth

Nachdem eine Anpassung des PLICs gezeigt wurde, welche sowohl eine globale Prioritätstreue, als auch die Unterbrechungsmigration unterstützt, soll nun, aufbauend auf der erweiterten PLIC Architektur, das InterSloth-Betriebssystem implementiert werden. Zunächst werden nun die Anforderungen und Ziele an InterSloth erläutert. Anschließend wird in Abschnitt 3.2.2 ein architekturieller Überblick geschaffen, welcher erläutert, welche Komponenten einen globalen Einplanungsansatz im Sloth-Betriebssystem möglich machen. Als Letztes werden in Abschnitt 3.3 diese Komponenten im Detail erklärt.

3.2.1 Designziele und Anforderungen

In weiten Teilen orientiert sich InterSloth an den Zielsetzungen der Sloth-Betriebssystemfamilie. Es soll also ein einheitlicher Prioritätenraum geschaffen werden, in dem Unterbrechungen und Tasks gleichartig behandelt werden, sodass das Problem der Prioritätsinversion implizit verhindert wird. Jeder Task ist eine Unterbrechung und jede Unterbrechung ist ein Task. Somit werden Einplanungsentscheidungen aus dem Betriebssystem ausgelagert und stattdessen im Unterbrechungskontroller getroffen. InterSloth soll dabei statisch und spezifisch für eine Anwendung und Hardwareplattform generiert werden, um eine möglichst enge Bindung zwischen Hardware und Applikation zu schaffen und die Geschwindigkeit und Größe des Systems signifikant zu verbessern.
Am Nächsten ist InterSloth dabei der MultiSloth-Variante, welche ebenfalls ein Multiprozessor-Sloth darstellt. Der wesentliche Unterschied zwischen InterSloth und MultiSloth besteht in der Art der Einplanung. Während MultiSloth einen rein partitionierten Einplanungsansatz implementiert, verfolgt InterSloth einen globalen Einplanungsansatz. Somit ist es nicht mehr nötig, Tasks in der Systemkonfiguration festen Prozessoren zuzuordnen, auf denen diese ausschließlich ausgeführt werden. Stattdessen werden Tasks dynamisch an die verfügbaren Prozessoren verteilt, je nach Auslastung und Priorität des Tasks und Prozessors. Insgesamt ist somit die Last des Gesamtsystems besser verteilt und es kann eine höhere Auslastung erreicht werden.
Weiterhin soll InterSloth Unterbrechungsmigration unterstützen. Hierbei wird ein verdrängter Task automatisch erneut angefordert und kann, nach der Priorisierung durch den Unterbrechungskontroller, auf einem anderen Prozessor fortgesetzt werden. Die Migration sollte hierbei für den verdrängten Task vollkommen transparent geschehen, sodass die Erweiterung des Einplanungsansatzes unbemerkt für den Anwendungsentwickler bleibt und keine Portierung bestehender Anwendungen erfordert. Gleichzeitig soll dabei die globale Prioritätstreue der MIRQ-V Semantik ausgenutzt werden um zu garantieren, dass der höchstpriore Task dem, zum Zeitpunkt der Aktivierung, niederpriorsten Kern zugeordnet wird. Globale Prioritätstreue garantiert nicht nur ein optimale Auslastung des Systems, sondern führt ebenfalls implizit zu der Verhinderung von Prioritätsinversionen. Zu jedem Zeitpunkt entspricht die Liste aus aktuell laufenden Tasks der Liste von den höchstprioren aktivierten Tasks . Im Gegensatz zu InterSloth gibt MultiSloth diese Garantie nur lokal – also pro Prozessor – und nicht global.
Bis auf diesen Unterschied soll InterSloth sich weitestgehend an MultiSloth orientieren. Es imple-

mentiert ebenfalls, wenn auch nur zu einem kleinen Teil, die AUTOSAR Spezifikation und erhält auf diese Weise die originale Sloth-Semantik. InterSloth implementiert aus der AUTOSAR Spezifikation nur Tasks. Bestehende Sloth-Anwendungen müssen somit nicht portiert werden, sofern sie keine Ressourcen oder Alarme besitzen.

Alarme können, aufgrund der Struktur des Unterbrechungssystems in RISC-V, nicht ohne grundlegende Modifizierung des Unterbrechungssystems implementiert werden. Timerunterbrechungen gehören unter RISC-V zur Klasse der lokalen Unterbrechungen, welche ausschließlich durch den CLINT verwaltet werden. Diese können im Gegensatz zu globalen Unterbrechungen nicht in den Priorisierungsprozess des PLICs einbezogen werden und können somit, im Gegensatz zur Sloth-Semantik, auch nicht im selben Prioritätenraum wie Tasks verwaltet werden.

Ressourcen können nicht rein durch das aus Abschnitt 2.1 bekannte *Priority Ceiling Protocol* verwaltet werden und erfordern ein globales Synchronisierungsprotokoll, da ansonsten einer gleichzeitige Inbesitznahme der gleichen Ressource auf verschiedenen Kernen nicht verhindert werden kann. Das in MultiSloth verwendete Synchronisierungsprotokoll MPCP (siehe [Mül+14]) basiert jedoch auf der Annahme, dass Tasks an einen Prozessor gebunden sind [Raj90]. Es ist somit nicht für einen globalen Einplanungsansatz geeignet und es muss ein alternatives Protokoll gefunden werden, welches sowohl eine Synchronisierung garantiert, als auch eine Verhinderung von Prioritätsinversionen. Es existieren Protokolle, welche eine Synchronisierung in Systemen mit globalen Einplanung garantieren (z.B. [Afs+15]), jedoch basieren diese auf Spinlocks und garantieren somit keine Verhinderung von Prioritätsinversionen. Dem Verfasser dieser Arbeit ist keine Arbeit bekannt, welche sowohl eine globale Synchronisierung, als auch die Verhinderung von Prioritätsinversionen garantiert.

Nachdem die Ziele und Anforderung an Sloth erläutert wurde, soll nun ein detaillierter Überblick über die zentralen Teile von InterSloth gegeben werden.

3.2.2 Architekturüberblick

Abbildung 3.3 zeigt eine Übersicht der InterSloth-Architektur. Im Vergleich zu der MultiSloth-Architektur Abbildung 2.6 fällt sofort ein entscheidender Unterschied auf:

Anstelle von einer Unterbrechungsroutinentabelle pro Kern, existiert nun eine gemeinsame Unterbrechungsroutinentabelle, welche jeder Prozessorkern (an dieser Stelle in RISC-V-Semantik „*Hart*“) zur Auswahl des auszuführenden Tasks benutzt (① in Abbildung 3.3). Diese wird, analog zu anderen Sloth-Varianten, statisch zur Übersetzungszeit aus der Systemkonfiguration generiert und kann somit zur Laufzeit nicht modifiziert werden. Durch den rein lesenden Zugriff auf die geteilte Datenstruktur und ihre Unveränderlichkeit ist an dieser Stelle keine weitere Synchronisation nötig und es entsteht keine zusätzliche Betriebslast durch ihre Teilung zwischen den verschiedenen Ausführungskontexten.

Weiterhin existiert nun zusätzlich eine Liste mit gesicherten Taskkontexten (② in Abbildung 3.3). Jeder Eintrag enthält neben einem Stapelzeiger und dem gesicherten Befehlszähler ebenfalls Registerwerte zum Zeitpunkt der Verdrängung.

Während sich in Sloth und SleepySloth *Basic Tasks* sich einen Stapel teilen können und dies in MultiSloth zumindest noch alle *Basic Tasks* auf einem Prozessorkern können, ist eine solche Einsparung von Stapelspeicher in InterSloth nicht mehr möglich. Ein Task kann jederzeit – vollkommen transparent – auf einem Prozessor durch einen höherprioren Task verdrängt, auf einen anderen Prozessor migriert und dort weiter ausgeführt werden. Zum Zeitpunkt der Verdrängung können also noch Werte des verdrängte Tasks auf dem Stapel liegen, welche gesichert werden müssen, um eine Fortsetzung dessen zu ermöglichen. Somit benötigt jeder Task, sowohl *Basic Tasks* als auch *Extended Tasks*, einen eigenen Stapelspeicherbereich, welcher ebenfalls zur Übersetzungszeit aus der Systemkonfiguration generiert wird.

Die Grenze zwischen *Basic Tasks* und *Extended Tasks* verschwimmt, denn durch die Tatsache das *Basic*

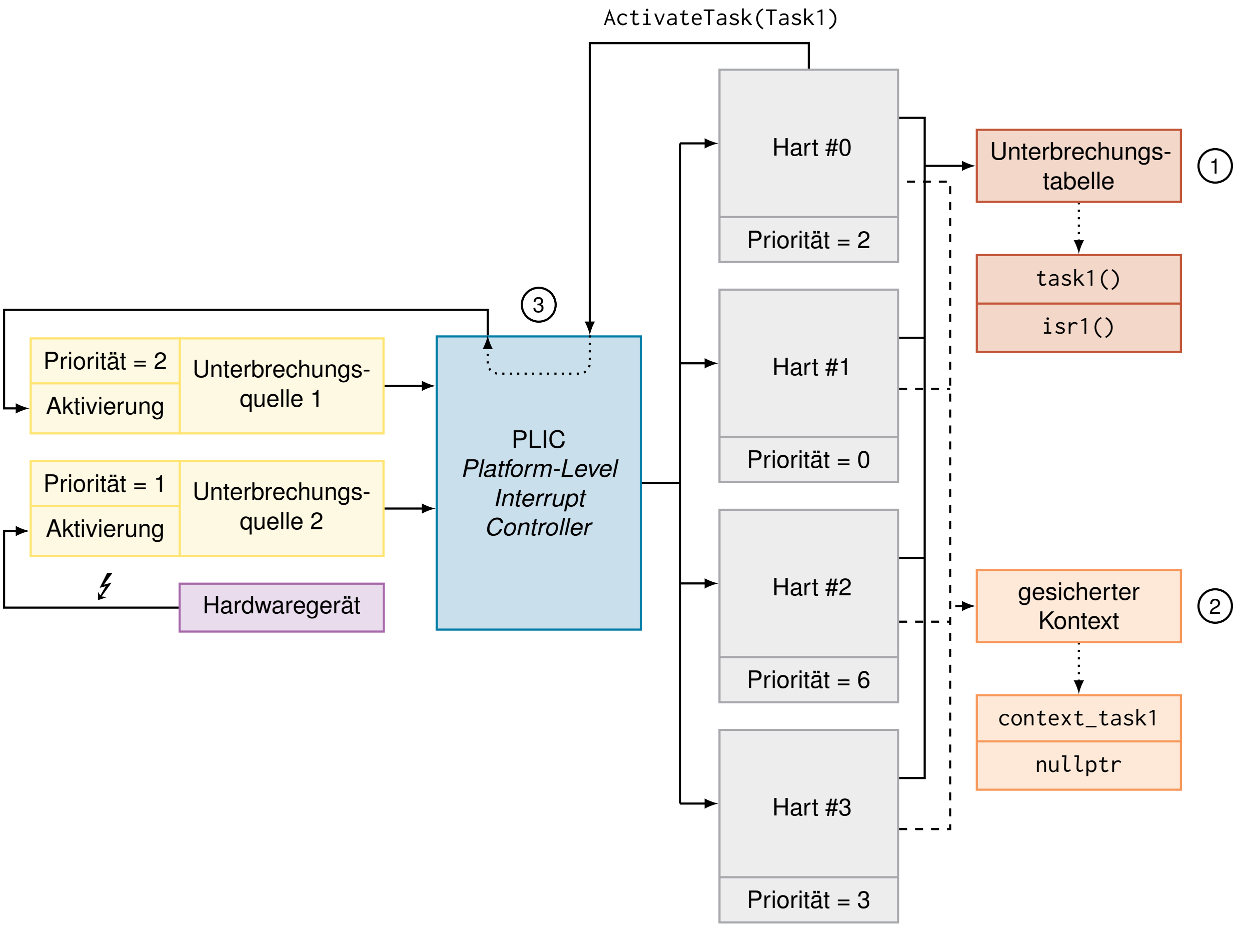


Abbildung 3.3 – Struktur des InterSloth-Betriebssystems

Tasks nun ihren eigenen Ausführungskontext besitzen, spricht nichts dagegen diesen auch Zugriff auf das Ereignisinterface (`SetEvent()`/`WaitEvent()`) zu gewähren. Der Unterschied zwischen *Basic Tasks* und *Extended Tasks* ist somit in InterSloth nur noch rein semantisch und hat keinen funktionellen Hintergrund mehr.

Außerdem geschieht sämtliche Tasklogik, d.h. Aktivierung, Terminierung oder Migration, durch Kommunikation mit dem PLIC, welcher seinerseits die entsprechenden Unterbrechungsquellen anpasst (③ Abbildung 3.3). Dies steht im Gegensatz zu der direkten Konfiguration der Unterbrechungsquellen in anderen Sloth-Varianten. Der PLIC enthält, wie in Abschnitt 3.1 beschrieben, Funktionen zur Aktivierung, Blockierung und Migrierung von Tasks. Er nimmt somit, im Gegensatz zu den anderen Sloth-Varianten, eine aktivere Rolle in der Taskverwaltung ein.

Das Kernstück des InterSloth-Betriebssystems stellt der geteilte Unterbrechungsbehandler dar. Die RISC-V-Architektur kann zum Zeitpunkt der Unterbrechung nicht zwischen verschiedenen globalen Unterbrechungen – also durch den PLIC verwaltete Unterbrechungen (siehe Abschnitt 2.3.2) – unterscheiden. Das aus SleepySloth und MultiSloth bekannte Konzept der statisch generierten Taskprologe ist somit nicht übertragbar auf die RISC-V-Architektur.

Es wurde daher entschieden, die Einlastung von Tasks in eine, zwischen allen Tasks geteilte, Un-

terbrechungsroutine auszulagern. Diese emuliert somit einen Vektorunterbrechungsmodus für ausschließlich globale Unterbrechungen.
Neben der Einlastung ist die globale Unterbrechungsbehandlungsroutine ebenfalls für die Kontextsicherung des unterbrochenen Tasks und die (eventuelle) Kontextwiederherstellung des aktivierten Tasks zuständig.

3.3 Implementierungsdetails

Nachdem ein grober Überblick über die Struktur des InterSloth-Betriebssystems gegeben wurde, soll nun genauer auf die einzelnen Teile eingegangen werden. Hierfür wird zunächst der Einlastungsmechanismus genauer vorgestellt, um anschließend auf die Implementierungen der Funktionen zur Taskaktivierung und -terminierung genauer einzugehen.

3.3.1 Einlastungsmechanismus

Abbildung 3.4 zeigt den Ablaufgraphen der globalen Unterbrechungsbehandlungsroutine. Es sei im

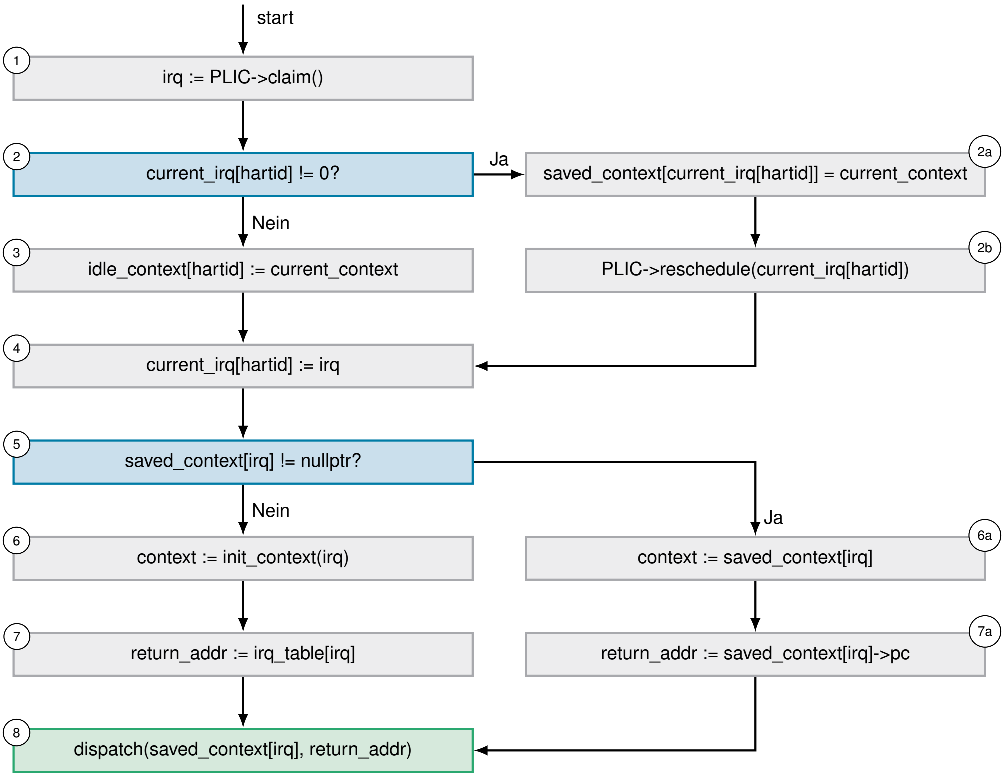


Abbildung 3.4 – Ablaufgraph für geteilte Unterbrechungsroutine des InterSloth-Betriebssystems.`current_context` enthält den Kontext des unterbrochenen Tasks, `hartid` die Hart ID des aktuellen Harts und `current_irq[]` eine Liste aus aktuell laufenden Unterbrechungen pro Hart.

Folgenden angenommen, dass `current_context` den Kontext des unterbrochenen Tasks enthält, `hartid` der Hart-ID des aktuellen Harts entspricht und `current_irq[]` eine Liste der aktuell laufenden Unterbrechung pro Hart ist.

Die Unterbrechungsbehandlungsroutine lässt sich in zwei Teile einteilen: Kontextsicherung des unterbrochenen Tasks und Kontextwiederherstellung des aktivierten Tasks.

Zu Beginn der Unterbrechungsbehandlungsroutine wird zunächst die Unterbrechung vom PLIC angenommen (*claim*), hierbei gibt der PLIC die **aktuell** dem Hart zugeteilte Unterbrechung zurück (① in Abbildung 3.4) Anschließend wird geprüft, ob gerade ein laufender Task unterbrochen wurde (②), d.h. momentan eine Unterbrechung behandelt wird. Falls ja, wird zunächst der Kontext des gerade unterbrochenen Task in einer globalen Kontextliste (`saved_context`) abgelegt ((2a)). Nun kann die zum unterbrochenen Task zugehörige Unterbrechung auf einen anderen Hart migriert werden ((2b)). Wurde kein Task unterbrochen, so wurde der Leerlauftask unterbrochen, dessen Kontext separat gesichert wird (③). Als Letztes wird die angenommene Unterbrechung gesichert (④). Damit ist nach Schritt (④) die Kontextsicherung abgeschlossen. Als nächstes muss der Kontext des aktivierten Task wiederhergestellt werden. Hierfür wird zunächst überprüft ob der zu aktivierende Task kürzlich unvollständig ausgeführt wurde, d.h. einen gesicherten Kontext besitzt (⑤). Besitzt der zu aktivierende Tasks einen zuvor gesicherten Kontext, so wird sich dieser gemerkt ((6a)) und der gesicherte Programmzähler als Rücksprungadresse extrahiert ((7a)). Alternativ wird zunächst ein initialer Kontext erstellt (⑥) und anschließend die Rücksprungadresse auf die durch den Sloth-Generator bereitgestellte Task-Funktion festgelegt (⑦).

Als Letztes wird schließlich, durch den Aufruf von `dispatch()`, der Kontext in die Register wiederhergestellt (⑧). Hierbei wird die gewünschte Rücksprungadresse in das CSR `mepc` geschrieben, auf welches der Programmzähler bei der anschließenden Ausführung der *Return-From-Interrupt*-Instruktion `mret` gesetzt wird. Nach der Ausführung der `mret` Instruktion, läuft somit entweder der neu aktivierte Task oder aber seine Fortsetzung sofern er zuvor unterbrochen wurde.

3.3.2 Taskaktivierung

Nachdem der Einlastungsmechanismus des InterSloth-Betriebssystems vorgestellt wurde, soll nun die Implementierung der Taskaktivierung im Detail erklärt werden.

Die Taskaktivierung in InterSloth funktioniert analog zu anderen Sloth-Implementationen. Zunächst werden Unterbrechungen auf dem auslösenden Hart gesperrt. Nun wird die zum Task gehörenden Unterbrechungsquelle – mit Hilfe des PLIC – ausgelöst, wodurch dieser, nach einer Priorisierung durch den PLIC, auf einen Hart zugestellt und dort ausgeführt wird.

Anschließend werden Unterbrechungen wieder eingeschaltet. Um die *synchrone* Semantik des `ActivateTask()` Systemaufrufes zu garantieren, müsste vor der Reaktivierung der Unterbrechungen noch ein feste Anzahl an Takten gewartet werden (siehe Abschnitt 2.2). Auf diese Weise ist sichergestellt, dass die Priorisierung aller Unterbrechungen im PLIC abgeschlossen ist, wenn die Unterbrechungen reaktiviert werden, und somit, sofern der neu aktivierte Task auf dem auslösenden Hart ausgeführt werden soll, die Reaktivierung den Einlastungspunkt des neu aktivierten Tasks markiert. Durch eine fehlende Hardwareimplementierung des modifizierten PLICs ist jedoch die Messung, wie viele Takte gewartet werden muss, nicht möglich. Aus diesem Grund handelt es sich streng genommen bei der in InterSloth implementierten `ActivateTask()` Variante nicht um einen *synchronen* Systemaufruf.

3.3.3 Taskterminierung und -chaining

Die Implementation der Taskterminierung (`TerminateTask()` Systemaufruf) hat große Ähnlichkeit zur Einlastungsroutine.

Zunächst werden wieder die Unterbrechungen gesperrt, um eine Verdrängung während des Systemaufrufes zu verhindern. Anschließend wird der Kontext des aktuell laufenden Tasks gelöscht, denn nach der Terminierung ist er in jedem Fall nicht mehr valide. Nun wird die zum Task zugehörige Unterbrechung im PLIC als *complete* markiert, wodurch der Task erneut aktiviert werden kann. Schließlich folgt eine Wiederherstellung des Leerlauftaskkontextes und eine anschließende Einlastung dessen. Hierbei wird die aus der Einlastungroutine bekannte `dispatch()`-Funktion aufgerufen, welche die zuvor deaktivierten Unterbrechung durch Ausführen der `mret` Instruktion wieder aktiviert.

Die Implementierung des *Task Chainings* (`ChainTask()` Systemaufruf) funktioniert analog zu `TerminateTask()`. Hierbei wird jedoch vor dem Aufruf der `dispatch()`-Funktion die zum Task, welcher durch den `ChainTask()` Aufruf als nächstes ausgeführt werden soll, gehörige Unterbrechung ausgelöst. Diese wird daraufhin durch den PLIC priorisiert und schließlich an einen Hart zugestellt. Analog zu `ActivateTask()` wäre an dieser Stelle eigentlich ein fester Wartezeitraum nötig um eine *synchrone* Semantik zu garantieren. Es handelt sich somit streng genommen bei in der InterSloth verwendeten `ChainTask()`-Implementation ebenfalls nicht um einen synchronen Systemaufruf.

3.3.4 Zusammenfassung

In diesem Kapitel wurde zunächst erklärt, welche Modifizierungen an der RISC-V-Architektur nötig sind um einen globalen Einplanungsansatz und Unterbrechungsmigration zu unterstützen. Anschließend wurde der so modifizierte PLIC genutzt, um eine Variante des Sloth-Betriebssystems – InterSloth – zu konstruieren, welche einen globalen Einplanungsansatz verfolgt. Auf diese Weise kann eine weitaus bessere Auslastung des Gesamtsystems im Gegensatz zu partitionierten Echtzeitbetriebssystemen erreicht werden.

Zunächst wurde hierfür ein grober Überblick über die Architekturunterschiede von InterSloth im Vergleich zu anderen Sloth-Varianten, insbesondere MultiSloth, gegeben. Im letzten Abschnitt wurden dann die Einlastungslogik, sowie die Implementierung der verschiedenen Tasksystemaufrufe besprochen, welche den Kern von InterSloth ausmachen.

ANALYSE

4

Ausgestattet mit dem Wissen über die Architektur von InterSloth und den modifizierten PLIC soll nun eine Evaluation von InterSloth folgen. Zunächst wird grob der Emulator/Virtualisierer QEMU eingeführt, welcher die Grundlage für die Evaluationsinfrastruktur bildet. Diese soll anschließend im Detail besprochen werden, um schlussendlich auf die verschiedenen Testfälle einzugehen. Hierbei soll zum einen funktionelle Korrektheit überprüft werden, d.h. ob eine strikte Prioritätstreue global eingehalten wird, und zum anderen die schon aus MultiSloth bekannte Latenzen zur Aktivierung und Terminierung von Tasks gemessen werden.

4.1 Evaluationsinfrastruktur

Die Evaluationsinfrastruktur setzt sich aus zwei Teilen zusammen. Das durch InterSloth generierte Betriebssystem wird zunächst durch eine modifizierte Variante des Systememulators QEMU ausgeführt. Auf diese Weise wird eine Logdatei erstellt, welche Nachrichten des modifizierten PLIC und des ausgeführten Programmcodes auf jedem Hart enthält. Die Logdatei wird anschließend durch ein Python-Skript ausgewertet um auf diese Weise die Korrektheit des Betriebssystems zu verifizieren. Weiterhin soll es verschiedene Geschwindigkeitsmetriken ermitteln, um einen Vergleich mit anderen Sloth-Varianten möglich zu machen.

4.1.1 QEMU

QEMU ist eine Multiplattformemulationsoftware, welche mit Hilfe von dynamischer Übersetzung eine sehr gute Emulationsgeschwindkeit erreicht [QEM18]. Hierfür werden zur Laufzeit Instruktionen des *emulierten* Prozessors (Gast) in Instruktion des *emulierenden* Prozessors (Host) übersetzt. Neben einer rein softwarebasierten Emulation, unterstützt QEMU auch verschiedene hardwarebasierten Verfahren zur Emulationsbeschleunigung wie KVM oder Xen.

QEMU ist weitestgehend unter der quelloffenen *GPL2* Lizenz veröffentlicht und ist auf den gebräuchlichsten Plattformen (GNU/Linux, macOS, Windows sowie BSD) lauffähig. Es kann über zehn verschiedene Architekturen (unter anderem IA32, amd64, PowerPC und ARM), sowohl in der Uni- als auch der Mehrprozessorvariante emulieren, wobei häufig verschiedene Implementierungen einer Architektur emulierbar sind. Eine Emulationsunterstützung für die RISC-V-Architektur befindet sich momentan in der Entwicklung.

QEMU kann Systeme auf zwei verschiedene Arten emulieren. Die sogenannte *User Mode Emulation* kann für ausgewählte Betriebssysteme Systemaufrufe emulieren, um so Userspace-Programme, welche für ein anderes Betriebssystem kompiliert wurden, ohne vollständige Hardwareemulation auszuführen. Die sogenannte *Full System Emulation* emuliert hingegen das gesamte System, d.h.

neben dem Prozessor können ebenfalls Peripheriegeräte wie Speicherkontroller, Unterbrechungskontroller, USB Kontroller sowie PS/2 Tastatur und Maus und eine VGA Grafikkarte emuliert werden.
Die Übersetzung von Instruktionen geschieht in Blöcken (sogenannten Translation Blocks (TBs)), welche dynamisch generiert werden und sich jeweils bis zu einer Sprunginstruktion erstrecken. Zusätzlich wird in jedem TB der Zustand der virtuellen CPU gesichert, wobei garantiert ist, dass dieser innerhalb des TBs nicht verändert wird. Sie werden anschließend in eine Zwischendarstellung überführt, welche von QEMU stark optimiert wird. Als Letztes werden sie durch die verschiedenen Architektur-Backends in native Instruktion übersetzt und ausgeführt. Hierbei können TBs, welche unkonditionell ineinander übergehen, zusammengefasst werden und auf einmal ausgeführt werden. Zusätzlich werden einmal ausgeführte TBs in einem Zwischenspeicher verwaltet, um so Programme mit hoher Lokalität zu beschleunigen. Durch diese dreigeteilte Übersetzungstrategie ist der als Tiny-Code Generator (TCG) bekannte dynamische Übersetzer von QEMU sehr schnell und kann trotzdem einfach an neue Architekturen angepasst werden. Bei Auftreten einer Ausnahme, kann QEMU mit Hilfe der zuvor übersetzten Translationblocks zurückverfolgen an welcher Instruktion die Ausnahme enstanden ist. Jedoch geht durch die blockbasierte Übersetzung die Möglichkeit verloren, genau festzustellen wann einzelne Instruktionen der emulierten Software ausgeführt werden.
Der in QEMU verwendete TCG erlaubt es weiterhin durch eine Startoption, jeden ausgeführten TB samt Adresse, Symbol und ausführenden Prozessor in eine Log Datei zu schreiben. Diese Funktionalität erlaubt eine primitive Analyse des ausgeführten Programms und wird in Abschnitt 4.1.3 genauer erläutert.

4.1.2 Anpassung von QEMU

Die Implementierung des RISC-V-Befehlssatzes in QEMU unterstützt zur Zeit vier verschiedene RISC-V-Architekturen:

- SiFive E Freedom SDK
- SiFive U Freedom SDK
- VirtIO
- Spike-Emulator kompatible Implementierung

Hierbei unterstützen nur die SiFive und die VirtIO Implementierungen überhaupt den PLIC und nur die VirtIO Implementierung erlaubt es Multiprozessorsysteme zu emulieren. Die Implementierung des PLICs in der VirtIO Implementierung ist dieselbe, welche auch in den SiFive Implementationen benutzt werden, wodurch sie die Speicherkarte des implementierten PLICs an die technische Beschreibung von SiFive hält. Aus diesem Grund wurde zum Testen des InterSloth-Betriebssystem für die VirtIO Implementierung der RISC-V-Emulation ausgewählt.
Zur vollständigen Emulation des InterSloth-Betriebssystem musste jedoch zunächst die Implementation des PLICs modifiziert werden. Wie in Abschnitt 3.1 beschrieben wurde hierfür als Erstes der interne Zustand von Unterbrechungen um einen weiteren Zustand (*Zugestellt*, siehe Abbildung 3.1) erweitert, um Wettlauf-Situationen bei einer Prioritätsänderung der CPU zu vermeiden. Weiterhin wurde Semantik des `claim/complete` Registers um die Möglichkeit von Unterbrechungsoperationen erweitert. Zusätzlich zur Beendigung einer Unterbrechung mit Hilfe des `claim/complete` Registers, besteht durch Schreiben des Registers die Möglichkeit Unterbrechungen Auszulösen und als Out-Of-Service zu markieren, um so eine erneute Zustellung an ein anderes Ziel anzufordern.
Als Letztes wurde die Priorisierungslogik des PLICs angepasst. Hierbei werden zunächst alle ausstehenden Unterbrechungen absteigend nach ihrer Priorität sortiert und anschließend die Liste

aus Harts aufsteigend nach ihrer Priorität sortiert. Die Priorität eines Harts entspricht dabei dem Maximum aus seiner Priorität und der Priorität der bereits zugestellten aber noch nicht angenommenen Unterbrechung (*claim* der Unterbrechung). So kann vermieden werden, dass eine niederpriore Unterbrechung fälschlicherweise an einen Hart zugestellt wird, dem bereits eine höherpriore Unterbrechung zugestellt wurde, welche er aber noch nicht angenommen hat. Dies kann passieren, da die Priorität des Harts erst bei Annahme der Unterbrechung auf die Unterbrechungspriorität hochgesetzt wird. Anstelle alle Harts zu unterbrechen, wird nun außerdem nur noch der Hart unterbrochen, welcher die niedrigste Priorität besitzt, die kleiner als die Unterbrechungspriorität ist. Auf diese Weise wird eine globale Prioritätstreue erreicht.

Weiterhin wurde der modifizierte PLIC um Logausgaben erweitert, um so eine Analyse des internen Zustands durch das in Abschnitt 4.1.3 beschriebene Skript zu erlauben.

4.1.3 Analysemethodik

Wie in Abschnitt 4.1.1 und Abschnitt 4.1.2 beschrieben, kann QEMU, unter Angabe von bestimmten Startoptionen, eine sogenannte Tracedatei erstellen, welche Logausgaben von verschiedenen Subsystemen enthält. Zur Analyse des InterSloth-Betriebssystems wurde entschieden, lediglich Logausgaben des modifizierten Unterbrechungssystems, d.h. PLIC, und des TCG auszuwerten.

Abbildung 4.1 zeigt eine solche Tracedatei, wobei unwichtige Information zwecks der Übersichtlichkeit entfernt wurden und die Funktionsnamen bereits dekodiert wurden. Ausgaben des TCG beginnen hierbei stehst mit „Trace“ und enthalten (von links nach rechts) neben der ID des Harts, den Beginn des TBs, wie viele Instruktionen in dem übersetzten TB vorhanden sind und das zugehörige Symbol (sofern vorhanden). Ausgaben des PLICs beginnen stets mit „plic:“ und erlauben Rückschlüsse darüber, wohin eine Unterbrechung zugestellt wurde, welche Unterbrechungen ausgelöst sind, wer eine Unterbrechung angenommen bzw. beendet oder eine Neuzustellung angefordert hat.

Zur Auswertung wurde ein Python-Skript geschrieben, welches zunächst die auf diese Weise erstellte Tracedatei einliest und jeder Zeile mit Hilfe von regulären Ausdrücken eine Art von Ereignis zuordnet. Ein Ereignis beschreibt hier sowohl die Ausführung eines TBs als auch z.B. das Auslösen einer Unterbrechung aus der Software. Durch diese Abstraktion der Tracedatei ist eine einfachere Analyse möglich.

Zur Verifikation des Betriebssystem wird die so gewonnene Liste aus Ereignissen in einen Verifizierer übergeben. Dieser merkt sich als erstes welche Unterbrechungen zu jedem Zeitpunkt ausstehend sind. Hierfür werden sowohl Unterbrechungsauslöseereignisses, als auch Unterbrechungsbeendigung und -neuzustellungsanforderungsereignisses ausgewertet. Weiterhin werden ebenfalls Ausführungsereignisses, d.h. Ausführungen von TBs, ausgewertet. Hier wird überprüft, ob es sich bei dem ausgeführten Symbol um eine Taskfunktion handelt. Ist dies der Fall, so wird geprüft ob diese zu einem der aktuell der N höchstprioren Tasks gehört (wobei $N :=$ Anzahl an Harts im System). Zur Überprüfung dieser Bedingung wird die zuvor erstellte Liste aus gerade ausstehenden Unterbrechungen verwendet, um rauszufinden welche Tasks zum Überprüfungszeitpunkt lauffähig sind. Die Zuordnung zwischen Task und Unterbrechung wird hierbei durch den Sloth-Generator bei der Übersetzung der Anwendung erzeugt und kann in Form einer JSON Datei in das Python-Skript eingebunden werden. Die so überprüfte Invariante der strikten globalen Prioritätstreue entspricht nicht exakt der geforderten Bedingung: Es ist zwar sichergestellt, dass ein laufender Task zu den N-höchstprioren aktivierten Tasks gehört, jedoch ist die richtige Zuordnung, d.h. der höchstpriore Task läuft auf dem niedrigstprioren Kern, auf diese Weise nicht überprüft. Eine genauere Überprüfung der Prioritätstreue wird jedoch durch die blockbasierte Übersetzung von QEMU und der relativen Größe der geteilten Unterbrechungsrouting im Vergleich zu Taskfunktionen in vielen Anwendungskonfigurationen deutlich erschwert und ist daher nicht Bestandteil dieser Arbeit.

```
Trace 0: [00000000800004ae/5] archChainTask(unsigned char)
Trace 0: [00000000800022b4/2] PLIC::set_hart_priority(unsigned char, unsigned char)
plic: write cpu priority: hart0-M old=1 new=0
Trace 0: [00000000800004c0/30] archChainTask(unsigned char)
plic: write claim: hart 0 irq 1 op 2
plic: trigger irq 1/prio:1/hart:0
plic: irq 1/prio:1 -> hart 0/prio:0
plic: (hart 0-M) meip: 1
Trace 1: [000000008000236a/1] PLIC::claim(void)
Trace 1: [000000008000236e/9] PLIC::claim(void)
plic: claim irq 2/prio:2/hart:1
Trace 0: [00000000800001ec/33]
plic: read claim: hart1-M irq=2
Trace 1: [00000000800005e0/2] interrupt_handler
Trace 1: [00000000800005e8/12] interrupt_handler
Trace 1: [0000000080000614/8] interrupt_handler
Trace 1: [000000008000075c/6] interrupt_handler
Trace 1: [000000008000076e/42] interrupt_handler
Trace 0: [00000000800000c0/1]
Trace 0: [00000000800000c4/32]
Trace 0: [0000000080000144/2]
Trace 0: [000000008000014c/2]
Trace 0: [0000000080000154/2]
Trace 0: [00000000800005a4/5] interrupt_handler
Trace 1: [00000000800001ec/33]
Trace 1: [000000008000040a/6] functionTask2(void)
Trace 1: [0000000080000346/1] archEndTask(void)
Trace 1: [000000008000034a/17] archEndTask(void)
```

Abbildung 4.1 – Beispiel einer QEMU Tracedatei

Die Geschwindigkeitsevaluation des InterSloth-Betriebssystems basiert auf derselben Grundlage. Die aus der Tracedatei gewonnenen Ereignisse werden ebenfalls an einen Zustandsautomaten übergeben, welcher zunächst nur Ausführungsereignisses verarbeitet. Hierbei wird für jeden Hart gespeichert, wie viele Instruktionen er bereits seit Beginn der Tracedatei ausgeführt hat. Weiterhin enthält der Zustandsautomat eine Liste aus Messungstypen, welche jeweils einen Startereignistyp und einen Endereignistyp besitzen. Entspricht das gerade verarbeitete Ereignis, einem Startereignistyp der enthaltenen Messung, so wird eine Instanz der Messung erzeugt und gesichert. Zusätzlich wird eine Kopie der aktuellen Hart-Zustände, d.h. wie viele Instruktionen bereits verarbeitet wurden, innerhalb der nun gestarteten Messungsinstanz gesichert. Weiterhin wird für alle gerade aktiven Messungen überprüft, ob das gerade verarbeitete Ereignis diese Beendet und falls ja wiederum eine Kopie der aktuellen Hart-Zustände gesichert. Jeder Messungstyp enthält zusätzlich eine Funktion, um die ihm zugeordnete Metrik, d.h. das Ergebnis der Messung, zu berechnen. Diese bezieht sich, in den hier vorgestellten Fällen, stets auf die Anzahl an Instruktionen, die in zwischen Beginn und Ende der Messung, in einem Hart ausgeführt wurden. Anschließend werden abgeschlossene Messungen sortiert nach ihrem Typ gesichert, und es wird – nach der vollständigen Verarbeitung der Tracedatei – sowohl der Durchschnittliche Messwert, als auch das Maximum und Minimum der Messwerte

ausgegeben.
Zur Zeit existieren vier verschiedene Typen von Messung:

ActivateTask Misst wie viele Takte auf dem Hart, auf welchem der aktivierte Task ausgeführt werden soll, vergangen sind bis der aktivierte Task tatsächlich ausgeführt wurde.

TerminateTaskToIdle Misst wie viele Takte vom Aufruf von `TerminateTask()` bis zu der Einlastung des Leerlauftasks vergehen.

TerminateTaskToOther Misst wie viele Takte vom Aufruf von `TerminateTask()` bis zu der Einlastung eines anderen Tasks vergehen.

ChainTask Misst wie viele Takte auf dem Hart, welcher `ChainTask()` aufgerufen hat, vom Aufruf von `ChainTask()` bis zur Einlastung des aktivierten Tasks geschehen.

In diesem Abschnitt wurde also ein Verfahren vorgestellt, wie, mit Hilfe der Logausgaben von QEMU, eine einfache Geschwindigkeits- und Verifikationsevaluation des InterSloth-Betriebssystems durchgeführt werden kann. Im Anschluss sollen nun zwei Verifikationstestfälle vorgestellt werden, welche zunächst die korrekte Funktionalität des InterSloth-Betriebssystems zeigen, um schlussendlich die Geschwindigkeit zu evaluieren.

4.2 Verifikationstestfälle

Dieser Abschnitt beschäftigt sich mit zwei Testfällen, welche die Korrektheit des InterSloth-Betriebssystems überprüfen sollen. Es wird zunächst überprüft, ob das System eine Unterbrechungsmigration korrekt ausführt und zu jeder Zeit die strikte Prioritätstreue wahrt. Anschließend wird überprüft, ob das System unter Änderung der Priorität falsch zugestellte Unterbrechungen erfolgreich neu zustellt.
In jedem der Testfälle läuft InterSloth auf einem RISC-V-System der RV64I Architektur mit insgesamt vier Harts. Alle dargestellten Fälle wurden mit Hilfe des zuvor beschriebenen Verifikators überprüft und bestanden diesen. Die dargestellten Aktivitätsdiagramme wurden manuell aus der QEMU-Tracedatei erstellt.

4.2.1 Prioritätstreue und Unterbrechungsmigration

In diesem Testfall soll überprüft werden ob sich das InterSloth-Betriebssystem prioritätstreu verhält, d.h. ob stets auf dem niedrigstprioren Hart der höchstpriore Task ausgeführt wird. Weiterhin soll überprüft werden, ob – im Falle einer Überauslastung des Systems – eine Verdrängung des niedrigstprioren Tasks stattfindet und, ob bei der Terminierung eines Tasks eventuelle noch nicht abgearbeitete Tasks erneut ausgeführt werden. Es wird also eine Überbelastungssituation herbeigeführt, die zunächst zu einer Verdrängung eines bereits laufenden Tasks führt. Der so eingelastete Task beendet anschließend – durch Setzen einer globalen Variable – einen anderen Task, wodurch schlussendlich der zuvor verdrängte Task wieder eingelastet werden kann.
Genauer gesagt existieren also fünf verschiedene Tasks (Abschnitt 4.2.1). Task 1 besitzt eine Priorität von eins und wird automatisch beim Systemstart aktiviert. Er wird hierbei vom PLIC dem ersten Hart zugeordnet und anschließend dort ausgeführt. Nach einer kleinen Verzögerung aktiviert Task1 nun wiederum Task 2 mit einer Priorität von drei, welcher auf dem zweiten Hart ausgeführt wird. Dieser aktiviert wiederum Task 3 (Priorität vier), welcher auf dem dritten Hart ausgeführt. Auf dieselbe Art wird Task 4 (Priorität fünf) auf dem letzten noch leerlaufenden Hart ausgeführt. Es ist nötig, alle Harts auf diese Weise mit Tasks zu belegen, um eine Migrationsituation zu erzeugen. Ansonsten wird

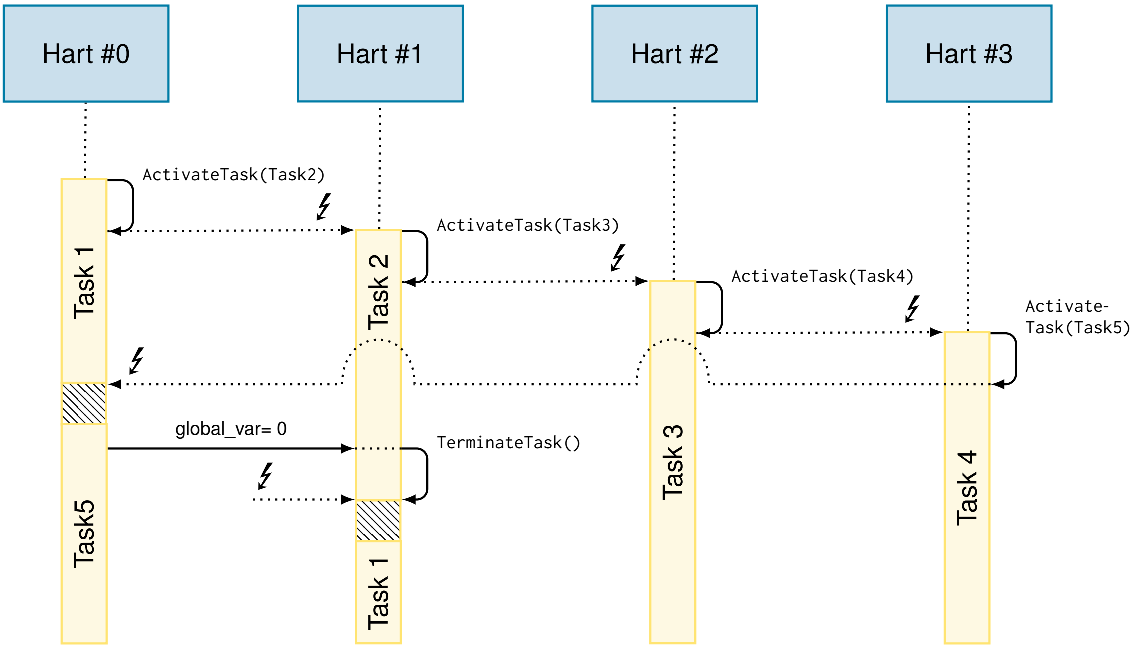


Abbildung 4.2 – Aktivitätsdiagramm für Verifikationtestfall zur Prioritätstreue und Unterbrechungsmigration. Gestrichelt dargestellt ist die Kontextsicherung und -wiederherstellung.

stets ein leerlaufender Hart für die Taskzuordnung bevorzugt, da er eine Priorität von null besitzt. Der nun von Task 4 aktivierte Task 5 (Priorität zwei) kann zunächst nicht trivial einem leerlaufenden Hart zugeordnet werden. Da aber seine Priorität (zwei) höher ist als die Priorität des Task 1 (eins), welcher gerade auf dem ersten Hart ausgeführt wird, wird Task 1 verdrängt und stattdessen Task 5 auf dem ersten Hart ausgeführt. Auf diese Weise bleibt die strikte Prioritätstreue erhalten. Task 1 kann durch seine niedrige Priorität jedoch keinen anderen Task verdrängen und verbleibt zunächst in einem wartenden Zustand. Nach einiger Zeit beendet der nun ausgeführte Task 5 durch Setzen einer globalen Variable Task 2. Daraufhin sinkt die Priorität des zweiten Harts auf null und der zuvor unterbrochene Task 1 kann erneut auf dem zweiten Hart ausgeführt werden. Der Kontext wird wiederhergestellt und Task 1 wird an der unterbrochenen Stelle fortgesetzt. InterSloth zeigt wie in Abschnitt 4.2.1 dargestellt das gewünschte Verhalten. Niederpriore Unterbrechungen werden migriert (sofern möglich) und es wird stets eine strikte Prioritätstreue gewahrt.

4.2.2 Prioritätsänderung

Nachdem in dem vorherigen Abschnitt ein allgemeiner Fall geprüft wurde und sichergestellt wurde, dass Unterbrechungen korrekt migriert werden, soll nun ein wichtiger Randfall verifiziert werden. Stellt der PLIC eine Unterbrechung an einen Hart zu, so wird das `claim/complete` Register mit dem Unterbrechungsindex gefüllt und der Hart unterbrochen. Anschließend liest dieser den Unterbrechungsindex aus dem `claim/complete` Register und bestätigt so die Bearbeitung der Unterbrechung. Wenn zwischen Zustellung und Bestätigung der Unterbrechung eine Prioritätsänderung eines Harts auftritt, so kann es passieren, dass der Hart, dem die Unterbrechung zugestellt wurde, nicht mehr länger das korrekte Ziel ist. In diesem Fall muss der PLIC, um die strikte Prioritätstreue zu erhalten, die Unterbrechung zurücknehmen und erneut an das richtige Ziel zustellen.

Um einen solchen Fall zu simulieren, werden erneut fünf Task benötigt. Zunächst wird das gleiche Szenario wie im vorherigen Testfall erzeugt, d.h. allen Harts wird ein Task zugeordnet. Der einzige Unterschied hierbei ist Task 2, welcher nun eine Priorität von zwei anstelle von drei besitzt. Somit hat wie zuvor beschrieben Hart 0 eine Priorität von eins, Hart 1 eine Priorität von zwei, Hart 2 eine Priorität von vier und Hart 3 eine Priorität von fünf. Anschließend deaktiviert Task 1, welcher auf Hart 0 ausgeführt wird, Unterbrechungen auf dem aktuellen Hart. Hierdurch wird sichergestellt, dass Hart 0 keine Unterbrechungen annehmen kann. Anschließend wird ein neuer Task mit Priorität drei aktiviert. Gemäß der strikten Prioritätstreue wird dieser vom PLIC an Hart 0 zugestellt, welcher jedoch gerade keine Unterbrechungen annehmen kann. Würde Hart 0 nun die Unterbrechungen erneut aktivieren, so würde er unterbrochen werden und anschließend der neu aktivierte Task ausgeführt werden. Nun setzt Hart 0 seine Priorität auf 6 hoch. Er ist somit kein valides Ziel mehr für den neu aktivierten Task, da seine Priorität größer ist als die des Tasks. Der PLIC muss nun die bereits zugestellte Unterbrechung zurücknehmen und an ein anderes Ziel zustellen. Hierfür wählt er das niedrigstpriore Ziel, also Hart 1, aus und unterbricht es anschließend.
Abschnitt 4.2.2 zeigt das Aktivitätsdiagramm des beschriebenen Falls. InterSloth verhält sich entsprechend der Erwartungen und stellt im Fall einer Prioritätsänderung die Unterbrechung an Hart 1 zu, wo, mit einer kurzen Verzögerung, der neu aktivierte Task ausgeführt wird.

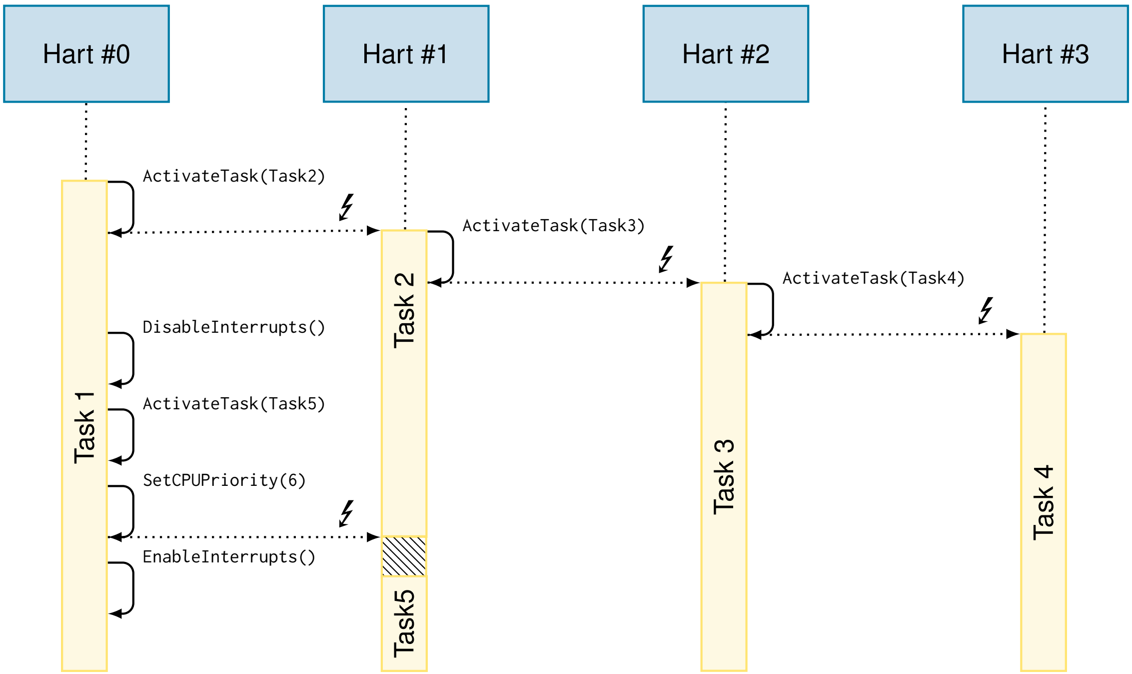


Abbildung 4.3 – Aktivitätsdiagramm für Verifikationstestfall zur Prioritätsänderung. Gestrichelt dargestellt ist die Kontextsicherung.

4.2.3 Zusammenfassung

Zusammenfassend zeigt InterSloth das gewünschte Verhalten. Es erhält in den gezeigten Fällen die strikte Prioritätstreue und migriert dafür gegebenenfalls niederpriore Tasks. Vor der Verdrängung wird der Taskkontext gesichert und nach der Migration wiederhergestellt, um eine reibungslose

Fortsetzung zu garantieren. Weiterhin verhindert es Wettlauf-Situationen im Falle einer Prioritätsänderung und nimmt falsch zugestellte Unterbrechungen erfolgreich zurück.

4.3 Geschwindigkeitsevaluation

Anschließend an die Verifikation des InterSloth-Betriebssystems soll nun die Geschwindigkeit evaluiert werden. Ein schneller und deterministischer Wechsel von Tasks ist essentiell für die Familie der Sloth-Betriebssysteme und soll daher auch in der InterSloth-Implementation erhalten bleiben. Zunächst wurde die Geschwindigkeit von Taskaktivierungen, d.h. entweder durch Aufruf von `ActivateTask()` oder `ChainTask()`, gemessen. Anschließend wurde noch die Terminierung, d.h. der Aufruf von *TerminateTask()* von Tasks evaluiert. Es wurden insgesamt drei verschiedene Mikrobenchmarks erstellt, um alle Ausgangsbedingungen für Taskwechsel abzudecken, welche im Folgenden vorgestellt werden sollen.

4.3.1 Taskaktivierung ohne Verdrängung

In diesem Testfall wurden insgesamt zwei Tasks konfiguriert. *Task1* wird automatisch beim Systemstart ausgeführt und startet in einer Schleife 10000-mal *Task2*. Dieser beendet sich sofort wieder, um dann erneut von *Task1* gestartet zu werden. Durch diesen Testfall können zwei Messung abgedeckt werden. Zum einen die Aktivierung eines Task auf einem Hart, welcher sich gerade im Leerlauf befindet, und zum anderen die Terminierung eines Task ohne nachfolgende Ausführung eines anderen Tasks. Abschnitt 4.3.1 zeigt die Ergebnisse dieses Mikrobenchmarks. `ActivateTask()` zeigt eine gute Geschwindigkeit von lediglich 166 Takten. Es ist somit schneller als das MultiSloth-Betriebssystem, welches für die gleiche Art von Taskaktivierung (Extended Task Aktivierung mit Stapelwechsel) im Durchschnitt 168 Takte benötigt. Im Gegensatz dazu ist die `TerminateTask()` Methoden mit 101 Takten bedeutend langsamer als ihr MultiSloth Äquivalent, welche nur 36 Takte benötigt. Dieser Geschwindigkeitsverlust kann jedoch einfach mit der zusätzlich Logik in der Taskterminierung erklärt werden. Im Gegensatz zu den bisherigen Sloth-Plattformen erfordert RISC-V zusätzlich die Rücksetzung des der CPU-Priorität und erfordert außerdem stets einen Stapelwechsel bei der Terminierung eines Tasks.

Übergang	InterSloth [Takte]	MultiSloth [Takte]
`ActivateTask()` ohne Verdrängung	166	168
`TerminateTask()` zu Leerlauf Task	101	36

Tabelle 4.1 – Ergebnisse des ersten Testfalls

4.3.2 Taskaktivierung mit Verdrängung

Dieser Testfall enthält insgesamt fünf verschiedene Tasks, von denen jedoch *Task3*, *Task4* und *Task5* lediglich Leerlauftasks darstellen. Beim Start des Systems wird zunächst nur *Task1* ausgeführt, welcher zunächst *Task3*,*Task4* und *Task5* startet. Auf diese Weise sind alle Harts im System mit Tasks belegt und besitzen eine Priorität, welche größer als null ist. Abschnitt 4.3.2 zeigt die Konfiguration der Harts nach dem Starten der Leerlauftasks. Um nun die Aktivierung eines Task bei Verdrängung eines anderen Tasks zu messen, aktiviert *Task1* *Task2* 10000-mal. Dieser besitzt eine Priorität von zwei und kann somit *Task3*, welcher auf Hart 1 ausgeführt wird, verdrängen. Der so unterbrochene

Task3 kann aufgrund seiner niedrigen Priorität (eins) zunächst nicht erneut zugeordnet werden und verbleibt in der Warteschlange des PLICs. Erst wenn sich anschließend *Task2* beendet, kann *Task3* erneut zugestellt werden und wird wieder ausgeführt. Weiterhin aktiviert *Task1 Task2* erst dann erneut wenn *Task3* wieder ausgeführt wird. Auf diese Weise kann neben der Aktivierung mit Verdrängung zusätzlich auch noch die Terminierung eines Tasks mit anschließender Einlastung eines anderen Tasks evaluiert werden. Abschnitt 4.3.2 zeigt die Ergebnisse des zweiten Testfalls. `ActivateTask()` zeigt weiterhin eine kompetitive Geschwindigkeit mit nur 166 Takten, welche exakt der Geschwindigkeit für eine Taskaktivierung ohne Verdrängung entspricht. Diese Ergebnis lässt durch die symmetrische Natur der Unterbrechungsroutine erklären. Sowohl bei der Einlastung vom Leerlauftasks als auch von einem anderen Tasks muss zunächst Kontext gesichert werden, da auch der Leerlauftask erneut fortgesetzt werden kann. Dies geschieht wenn ein Task sich beendet und kein neuer Task eingelastet werden kann. Es macht somit für das InterSloth-Betriebssystem keinen Unterschied, ob der Leerlauftask oder ein zuvor laufender Task verdrängt wird. Zur Terminierung eines Tasks mit anschließender Einlastung ergibt sich ein Ergebnis von 209 Takten. Aus den vorherigen Ergebnissen würde man jedoch zunächst ein Ergebnis von 267 Takten erwarten (101 Takte Terminierung und anschließend 166 Takte Aktivierung). Der Unterschied ergibt sich aus der Tatsache, dass der neu eingelastete Task eine Fortsetzung eines bisherigen Tasks ist und somit der Initialisierungsaufwand für die Kontextstrukturen in der Unterbrechungsbehandlungsroutine entfällt. Insgesamt ist die Terminierung eines Tasks jedoch immer noch langsamer als ihr MultiSloth-Äquivalent. Dies ist weiterhin mit der zusätzlichen Logik in der Terminierung zu erklären.

Hart ID	Priorität	ausgeführter Task
0	3	*Task1*
1	1	*Task3*
2	4	*Task4*
3	5	*Task5*

Tabelle 4.2 – Hartkonfiguration für Testfall 2 nach Starten der Leerlauftasks

Übergang	InterSloth [Takte]	MultiSloth [Takte]
`ActivateTask()` mit Verdrängung	166	168
`TerminateTask()` zu anderem Task	209	121

Tabelle 4.3 – Ergebnisse des zweiten Testfalls

4.3.3 Taskchaining

Abschließend soll nun die Geschwindigkeit des `ChainTask()`-Systemaufrufes evaluiert werden. Dies geschieht mit Hilfe von zwei verschiedenen Tasks, welche jeweils gegenseitig `ChainTask()` mit dem anderen Task als Parameter aufrufen. Analog zu den anderen Testfällen wird auch dieser Test nach 10000 Wiederholungen abgebrochen. Es entsteht ein System, in dem sich zwei Tasks ständig abwechseln. Eine Geschwindigkeitsmessung ergibt eine Geschwindigkeit zwischen 200 und 207 Takten für den `ChainTask()`-Systemaufruf. Die Schwankung lässt durch die ungenaue Messmethodik erklären. MultiSloth benötigt für denselben Aufruf lediglich 125 Takte, wobei dies jedoch keine Stapelwechsel beinhaltet. Die zusätzlichen 75 bzw. 82 Takte in InterSloth werden durch diesen verbraucht, da InterSloth keine Möglichkeit besitzt nicht den Stapel zu wechseln, wenn ein Task gewechselt wird.

4.4 Zusammenfassung

Zusammenfassend wurden alle wichtigen Taskübergänge evaluiert. Im Vergleich zu MultiSloth zeigt sich eine gute Geschwindigkeit. Insbesondere die Taskaktivierung ist in InterSloth sogar schneller als in MultiSloth. Andere Taskübergänge erreichen vergleichbare Geschwindigkeiten. Durch die ungenaue Messmethodik ist jedoch eine genaue Evaluation nur schlecht möglich. Die hier dargestellten Zahlen stellen somit nur eine Approximation der echten Geschwindigkeit dar. In weiterführenden Arbeiten soll InterSloth auf echter Hardware evaluiert werden, wodurch eine exaktere Messung möglich ist.

ZUSAMMENFASSUNG 5

Diese Arbeit zeigt die Implementierung eines globalen Einplanungsansatzes in einem Sloth-Betriebssystem. Verdrängte Tasks können auf einen anderen Kern migriert werden, und es ist stets gewährleistet, dass die höchstprioren Tasks auf den niedrigstprioren Kernen laufen. Auf diese Weise garantiert InterSloth eine strikte Prioritätstreue und verhindert, durch die geschickte Nutzung des Unterbrechungskontrollers, Prioritätsinversionen durch Unterbrechungen.

InterSloth basiert auf der Sloth-Betriebssystemfamilie und kann dadurch viele nicht-funktionale Eigenschaften dessen erhalten. Ebenso wie andere Sloth-Varianten wird es zur Übersetzungszeit an die Anwendung angepasst und statisch konfiguriert. Das so entstehende Betriebssystem ist minimal und erhöht neben der Geschwindigkeit ebenfalls die Verifizierbarkeit, welche für den Einsatz in harten Echtzeitsystemen häufig nötig ist. Um InterSloth zu implementieren, musste zunächst durch Portierung der MIRQ-V Funktionalität in eine existierende Architektur die Hardwarevorraussetzungen geschaffen werden. Als Zielplattform wurde hierfür die offene Befehlssatz-Plattform RISC-V gewählt und der zugehörige Unterbrechungskontroller PLIC im QEMU Systememulator um die gewünschte Funktionalität erweitert. Anschließend wurde der Sloth-Generator auf RISC-V portiert, um schlussendlich mit Hilfe des modifizierten Emulators, eine Sloth-Variante zu entwickeln, welche Tasks global einplant und somit eine bessere Auslastung des Gesamtsystems erzielen kann. In der Evaluation zeigte sich eine vergleichbare Geschwindigkeit zur partitionierten Multikern-Variante – MultiSloth – des Sloth-Betriebssystems. In Bezug auf Taskaktivierungen zeigt InterSloth sogar eine verbesserte Geschwindigkeit gegenüber MultiSloth.

Die Messungen der Geschwindigkeit geben, aufgrund der QEMU-basierten Messmethodik, jedoch nur approximativ die reale Geschwindigkeit wieder. Parallel zu dieser Arbeit findet die Portierung des MIRQ-V Unterbrechungskontrollers in die offene Implementierung der RISC-V Architektur – dem Rocket-Chip – als seperate Arbeit statt. Zukünftig könnte hierdurch eine genauere Evaluation des InterSloth-Betriebssystems möglich sein.

Weiterhin implementiert InterSloth zur Zeit nur ein kleines Subset der OSEK bzw. AUTOSAR OS Funktionalität. Zukünftige Arbeit am InterSloth-Projekt könnte weitere Subsysteme, wie Ereignisse oder Kommunikation zwischen verschiedenen Tasks, implementieren. Zusätzlich ist die Implementierung eines gemischten Einplanungsansatzes – mit Hilfe des Timersubsystems der RISC-V Architektur – theoretisch möglich und somit ebenfalls eine denkbare Erweiterung des InterSloth-Projekts.

ABKÜRZUNGSVERZEICHNIS

ISR	Interrupt-Service Routine
PLIC	Platform-Level Interrupt Controller
CLINT	Core-Local Interruptor
PC	Program Counter
CSR	Kontroll- und Statusregister
TCG	Tiny-Code Generator
TB	Translation Block
SDK	System-Development Kit
RISC	Reduced-Instruction Set Computer
OSEK	Offene Systeme und deren Schnittstellen für Elektronik im Kraftfahrzeug
SRN	Service-Request Node
ICU	Interrupt Control Unit
ABI	Application Binary Interface
MPCP	Multicore Priority Ceiling Protocol
PCP	Priority Ceiling Protocol
GPL	GNU General Public License
hart	HartHardware-Thread

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

QUELLCODEVERZEICHNIS

ALGORITHMENVERZEICHNIS

LITERATUR

[Afs+15] S. Afshar u. a. “Resource sharing in a hybrid partitioned/global scheduling framework for multiprocessors”. In: *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*. Sep. 2015, S. 1–10. DOI: `10.1109/ETFA.2015.7301456`.

[AP14] Krste Asanović und David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Techn. Ber. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, Aug. 2014. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html`.

[Asa16] Krste Asanović. Jan. 2016. URL: `https://riscv.org/wp-content/uploads/2016/01/Tues1000-RISCV-20160105-Updates.pdf` (besucht am 07. 08. 2018).

[AUT13] AUTOSAR. *Specification of Operating System (Version 5.1.0)*. Techn. Ber. Automotive Open System Architecture GbR, Feb. 2013.

[Bew16] Christian Bewermeyer. “Priority-Obedient Multicore Interrupt Controller”. Bachelor Thesis. Friedrich-Alexander University Erlangen-Nuremberg, Nov. 2016.

[Blo+07] A. Block u. a. “A Flexible Real-Time Locking Protocol for Multiprocessors”. In: *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*. Aug. 2007, S. 47–56. DOI: `10.1109/RTCSA.2007.8`.

[Bra11] Björn B. Brandenburg. “Scheduling and Locking in Multiprocessor Real-Time Operating Systems”. Diss. The University of North Carolina at Chapel Hill, 2011. URL: `http://www.cs.unc.edu/~bbb/diss/`.

[Dig17] Western Digital. Nov. 2017. URL: `https://www.wdc.com/about-wd/newsroom/press-room/2017-11-28-western-digital-to-accelerate-the-future-of-next-generation-computing-architectures-for-big-data-and-fast-data-environments.html` (besucht am 07. 08. 2018).

[Fou18a] RISC-V Foundation. 2018. URL: `https://riscv.org/faq/` (besucht am 07. 08. 2018).

[Fou18b] RISC-V Foundation. 2018. URL: `https://riscv.org/risc-v-cores/` (besucht am 07. 08. 2018).

[HLSP11] Wanja Hofer, Daniel Lohmann und Wolfgang Schröder-Preikschat. “Sleepy Sloth: Threads as Interrupts as Threads”. In: *Proceedings of the 32nd IEEE International Symposium on Real-Time Systems (RTSS ’11)*. (Vienna, Austria, 29. Nov.–2. Dez. 2011). IEEE Computer Society Press, Dez. 2011, S. 67–77. ISBN: 978-0-7695-4591-2. DOI: `10.1109/RTSS.2011.14`.

[Hof+09] Wanja Hofer u. a. “Sloth: Threads as Interrupts”. In: *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS ’09)*. (Washington, D.C., USA, 1.–4. Dez. 2009). IEEE Computer Society Press, Dez. 2009, S. 204–213. ISBN: 978-0-7695-3875-4. DOI: `10.1109/RTSS.2009.18`.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Second Edition. Springer-Verlag, 2011. ISBN: 978-1-4419-8236-0.

[Loh+09] Daniel Lohmann u. a. “CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems”. In: *Proceedings of the 2009 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, Juni 2009, S. 215–228. ISBN: 978-1-931971-68-3.

[Mül+14] Rainer Müller u. a. “MultiSloth: An Efficient Multi-Core RTOS using Hardware-Based Scheduling”. In: *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS ’14)*. (Madrid, Spain). Washington, DC, USA: IEEE Computer Society Press, 2014, S. 289–198. ISBN: 978-1-4799-5798-9. DOI: `10.1109/ECRTS.2014.30`.

[oA17] openrisc.io und Authors. *OpenRISC Specification Version 1.2*. Techn. Ber. Aug. 2017. URL: `https://github.com/openrisc/doc/blob/master/openrisc-arch-1.2-rev0.pdf` (besucht am 06. 07. 2018).

[Pö] Antje Pöckel. *Der DLX Prozessor*. Techn. Ber. URL: `https://web.archive.org/web/20140309025545/http://www.mr.inf.tu-dresden.de/studium/winter/infet1/docs/DLX.pdf` (besucht am 06. 07. 2018).

[Raj90] Ragunathan Rajkumar. “Real-time synchronization protocols for shared memory multiprocessors”. In: *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS ’90)*. Washington, DC, USA: IEEE Computer Society Press, 1990, S. 116–123. DOI: `10.1109/ICDCS.1990.89257`.

[WAI17a] Andrew Waterman, Krste Asanović und SiFive Inc. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. Techn. Ber. Mai 2017. URL: `https://riscv.org/specifications/` (besucht am 06. 07. 2018).

[WAI17b] Andrew Waterman, Krste Asanović und SiFive Inc. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. Techn. Ber. Mai 2017. URL: `https://riscv.org/specifications/privileged-isa/` (besucht am 06. 07. 2018).

[OSE05] OSEK/VDX Group. *Operating System Specification 2.2.3*. Techn. Ber. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2014-09-29. OSEK/VDX Group, Feb. 2005.

[QEM18] QEMU Documentation. 2018. URL: `https://qemu.weilnetz.de/doc/qemu-doc.html` (besucht am 17. 07. 2018).