hhu.

Bachelorarbeit

Entwicklung eines ARM-Kernels für hhuOS

vorgelegt von

Markus Schäfer

aus Gummersbach

Abteilung Betriebssysteme
Prof. Dr. Michael Schöttner
Heinrich-Heine-Universität Düsseldorf

11. Oktober 2023

Erstgutachter: Prof. Dr. Michael Schöttner
Zweitgutachter: Prof. Dr. Martin Mauve
Betreuer: M.Sc. Fabian Ruhland

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

70 Prozent der Weltbevölkerung benutzen ARM-basierte Produkte. [1] Dies zeigt, wie verbreitet die Architektur ist. ARM-basierte Produkte finden sich in den unterschiedlichsten Bereichen. Unter anderem sind sie in Smartphones, PCs, Server zu finden. Vor allem im Smartphone Bereich beträgt der Anteil an ARM-basierten Geräten 99 Prozent. [1] Auch in den Bereichen Server und PCs zeigt sich ein wachsender Anteil an ARM-basierten Produkten. So hat Supermicro, der weltweit drittgrößte Anbieter für Server [2], ARM-basierte Server in ihrer Produktpalette hinzugefügt. Eine der Gründe für dessen Verwendung ist, dass eine hohe Leistung pro Watt erreicht werden soll. [3] Im PC Bereich hat Apple bereits alle seine Produkte auf ARM-basierte Chips umgestellt. So wurde bereits 2020 angekündigt, dass ihre Laptops und iMacs von Intel auf Apple Chips, ARM basierte Chips, umgestellt werden. Als eine der Hauptgründe führen sie die höhere Leistung pro Watt an. Diese erlaubt Ihnen, im Vergleich zur ihrer vorherigen Intel-basierten Generation, eine doppelte Akkulaufzeit zu ermöglichen. [4] Apple hält auch 2023 weiter an diesem Vorgehen fest. Der letzte veröffentlichte Chip von Apple ist der M2 Ultra, ebenfalls ein ARM-basierter Chip, bei dem auch eine energieeffiziente Performance ein Hauptvorteil ist. [5] Dies hat zur Folge, dass im Laufe der Zeit alle Geräte auf welchen MacOs laufen, ARM-basierte Geräte sein werden. MacOs macht weltweit ungefähr 15 Prozent der Betriebssysteme aus. [6]

Aufgrund der zunehmenden Bedeutung der ARM-Architektur sollte auch dessen Bedeutsamkeit in der Lehre von zukünftigen Informatikern an Bedeutung gewinnen. In dem Masterkurs Betriebssystem-Entwicklung der Universität Düsseldorf soll der Student ein Betriebssystem für die x86-Architektur entwickeln. Dies war bisher ausreichend, da die meisten PC Prozessoren von Intel oder AMD stammen, welche nur x86 Prozessoren herstellen. [7] Im Zuge der steigenden Bedeutung der ARM-Architektur ist es sinnvoll neben dem x86 auch eine ARM-Lösung anzubieten. So bietet sich der Kurs als gute Basis für die Entwicklung eines ARM-basierten Betriebssystems an.

Aufgrund der oben dargelegten Inhalte ist das Ziel dieser Arbeit, ein Betriebssystem für die ARM-Architektur zu entwickeln und auf Basis des Masterkurses Betriebssystem-Entwicklung eine Lösung zu entwerfen. Dazu werden die einzelnen Aufgaben des Kurses für die ARM-Architektur bearbeitet. Zudem werden Unterschiede zur x86-Architektur

herausgearbeitet und analysiert. Das Betriebssystem soll für 64-Bit ARM A-Kerne entwickelt werden. Zum einen, da während des Kurses ebenfalls ein 64-Bit System in der Übung entwickelt wird und zum anderen, da die A-Serie die Serie ist, die für denselben Anwendungszweck wie x86 benutzt wird. Außerdem wird die aktuelle Architektur Version Armv8 verwendet. Anstelle echter Hardware wird in dieser Arbeit, wie im Kurs auch, die Virtualisierungssoftware Qemu verwendet. Diese emuliert die gesamte Hardware eines Computers. Ein Vorteil der Virtualisierungssoftware ist die gute Verfügbarkeit für Studenten. Spezifisch wird der ARM System Emulator "qemu-system-aarch64"mit der "generic virtual platformäls Maschine genutzt. Dies wird umgesetzt, um echte Hardware von einer spezifischen Maschine zu vermeiden, damit das System so allgemein wie möglich ist.

1.2 Aufbau der Arbeit

Im Kapitel Grundlagen werden die erforderlichen Kenntnisse für die Entwicklung eines Betriebssystems für die ARM-Architektur durchgegangen. Im Unterkapitel Die ARM-Architektur wird eine Übersicht über relevante Grundfunktionen der für diese Arbeit ausgewählten ARM-Architektur gegeben. Daraufhin wird im Unterkapitel Der Kurs ein Überblick über die Ziele und Inhalt des Masterkurses gegeben. Im Kapitel Der Kurs in ARM folgt die Implementation des Betriebssystems. Da die Implementation dem Kurs als Leitfaden folgen soll, wird jede Aufgabe als eigenes Unterkapitel behandelt. In diesen wird die Aufgabe in ARM realisiert. Des Weiteren werden Unterschiede zur x86-Architektur herausgearbeitet und analysiert. Im Kapitel Verification wird kurz erläutert, wie die Funktionalität des entwickelten Betriebssystems verifiziert wird. Abschließend werden im Kapitel Fazit die Ergebnisse dieser Arbeit zusammengefasst, Schlussfolgerungen gezogen und Ansatzpunkte für zukünftige Untersuchungen aufgezeigt.

Kapitel 2

Grundlagen

Die Entwicklung eines Betriebssystems für ARM erfordert Kenntnisse über Grundfunktionen der ARM-Architektur. Daher soll im Kapitel Die ARM-Architektur eine Übersicht über relevante Grundfunktionen, der für diese Arbeit ausgewählten ARM-Architektur, gegeben werden. Außerdem ist es notwendig, einen Überblick über die Ziele und Inhalte des Master Kurses zu erlangen, da auf Basis dieses Kurses das Betriebssystem entwickelt werden soll. Im Kapitel Der Kurs werden die Ziele und Inhalte des Masterkurs "Betriebssystementwicklung" erläutert.

2.1 Die ARM-Architektur

Die ARM-Architektur wurde ursprünglich 1983 von dem britischen Unternehmen Acorn entwickelt. Der Name ARM ist ein Akronym und steht für Acorn Risc Machine. 1990 hat die Firma Acorn zusammen mit Apple und VLSI Technology das Unternehmen Advanced RISC Machines Limited in Großbritannien gegründet. Die Entwicklung der ARM-Architekturen wurde in diese Firma ausgelagert und später in ARM Limited umbenannt. [8]. Die ARM-Architektur wird von der Firma ARM Limited entwickelt und an Halbleiter-Entwicklungsunternehmen sowie Halbleiterhersteller lizenziert. Die Firma ARM produziert keine Chips. [8] Die ARM-Architektur wird oftmals nur als ARM bezeichnet.

Die ARM-Architektur ist eine Familie von Befehlssätzen, welche auf Reduced Instruction Set Computing (RISC) Architektur basieren. Das bedeutet, dass ARM nicht eine Architektur ist, sondern eine Familie von Architekturen. Aus diesem Grund befindet sie sich in vielen unterschiedlichen Geräten. Von Microcontrollern in Kaffeemaschinen bis hin zu Servern. [9] Die Firma hat seit 1985 zahlreiche Änderungen an den Architekturen vorgenommen. Zur besseren Unterscheidbarkeit wurden diese Änderungen in Versionen unterteilt. Diese werden abgekürzt ARMv[Versionsnummer] genannt. Um die unterschiedlichen Anwendungszwecke zu bedienen, hat die Firma seit ARMv6 die Cortex-Architekturen eingeführt. Diese unterteilen die Architekturen in drei Hauptserien. Für den Microcontroller Bereich wird die M-Serie verwendet. Diese wird zum Beispiel in Kaffeemaschinen eingesetzt. Die R-Serie wird für real-time devices und die A-Serie für Applikations genutzt. Applikationen bezeichnen hier betriebssystembasierte Anwendungen wie Smartphones und Laptops. Im Jahr 2020 wurde die Serie A um

den Cortex X erweitert, dessen Kerne für Rechenleistung optimiert wurden. Zudem hat die Firma seit 2018 die Neoverse-Familie hinzugefügt. Diese Reihe findet Anwendung als Server-CPU. [8] Des Weiteren gibt es benutzerdefinierte Familien, welche auf ARM basieren. Bekannte Vertreter sind die Apple-CPUs und Nvidias Grace-CPUs. [9]

Aufgrund der vielen verschiedenen Versionen der ARM-Architektur und deren RISC-Basis ist Rückwärtskompatibilität keine Gegebenheit. Dies bedeutet, dass nicht alle Features einer Version in der nächsten Version verfügbar sein müssen. Grund dafür ist, dass versucht wird, nicht oder wenig benötigte Transistoren auf dem CPU Die zu vermeiden. Dies kann Platz auf dem CPU Die für wichtigere Komponenten frei machen oder schlichtweg Einsparungen in der Hitzeentwicklung bedeuten. [9] Im weiteren haben die Unterschiede zwischen den Versionen und den Familien zur Folge, dass ARM kein einheitliches Assembler hat. Befehle können sich unterscheiden oder für einen Kern gänzlich fehlen. [9]

Die Serie, die am nächsten an der x86-Architektur ist, ist die A-Serie. Wie in der Einleitung bereits festgelegt, wird die 64-Bit der A-Serie ARMv8 genutzt. Dementsprechend wenn in der Arbeit von der ARM-Architektur gesprochen wird, ist diese ausgewählte Architektur gemeint.

Im Folgenden werden die Grundlagen der Architektur, welche für die Entwicklung eines Betriebssystems in ARM notwendig sind, erläutert. Diese sind Berechtigungen, Exceptions, Register, Supervisor Call, Boot, Generic Interrupt Controller, Timer und serielle Schnittstelle.

2.1.1 Berechtigungen

Moderne Software wird in unterschiedliche Module unterteilt. Jedes Modul hat dabei ein anderes Zugriffslevel auf das System und seine Ressourcen. Eine solche Unterteilung findet sich auch zwischen dem Betriebssystem Kernel und den User Anwendungen. Diese beiden Module sollten unterschiedliche Zugriffsrechte haben, da User Anwendung nicht die Berechtigungen haben sollten, auf Systemressourcen direkt zuzugreifen, die das Verhalten des Kernel beeinflussen können. Im Gegensatz dazu benötigt der Kernel Berechtigungen, auf die nötigen Systemressourcen zugreifen zu können, um seine Aufgaben zu erledigen. Aus diesem Grund besitzt die ARM-Architektur unterschiedliche Ebenen von Berechtigungen, auch Privileged Levels genannt. Das Privilege Level gibt vor, auf welche Ressourcen des Prozessors die Software zugreifen kann. [10]

Die ARM-Architektur hat zwei Haupt-Berechtigungsebenen. Privileged Mode und Non-Privileged Mode. Der Non-privileged Mode wird auch als User Mode bezeichnet und ist aktiv, wenn der Kern sich nicht in einer Exception befindet. Darüber hinaus hat er die geringste Menge an Berechtigungen, besitzt seinen eigenen Stackpointe sowie Paging-Tabelle und Exception Vektor Tabelle. Sollte der Kern sich in einer Exception befinden, so befindet er sich im Privileged Mode und hatte damit Zugriff auf alle Ressourcen des Prozessors. [9] Das heißt die einzige Möglichkeit vom "Non-privileged Modeïn den "Privileged Modeßu gelangen, ist durch das Eintreten in eine Exception.

Sollte die TrustZone Security Extension implementiert sein, gibt es drei Exception Levels. Alle drei Exception Level arbeiten im "Privileged Mode", haben jedoch unterschiedliche Berechtigungen. Wobei der höchste Exception Level alle Berechtigungen hat und mit abnehmenden Level der Zugriff auf Ressourcen eingeschränkt wird. Wie auch im Non-Privileged besitzt jedes Exception Level seinen eigenen Stack Pointer, Paging Tabelle und Exception Vektor Tabelle. Die Aufteilung selbst erfolgt nach Zweck bzw. Aufgabe der einzelnen Ebenen.

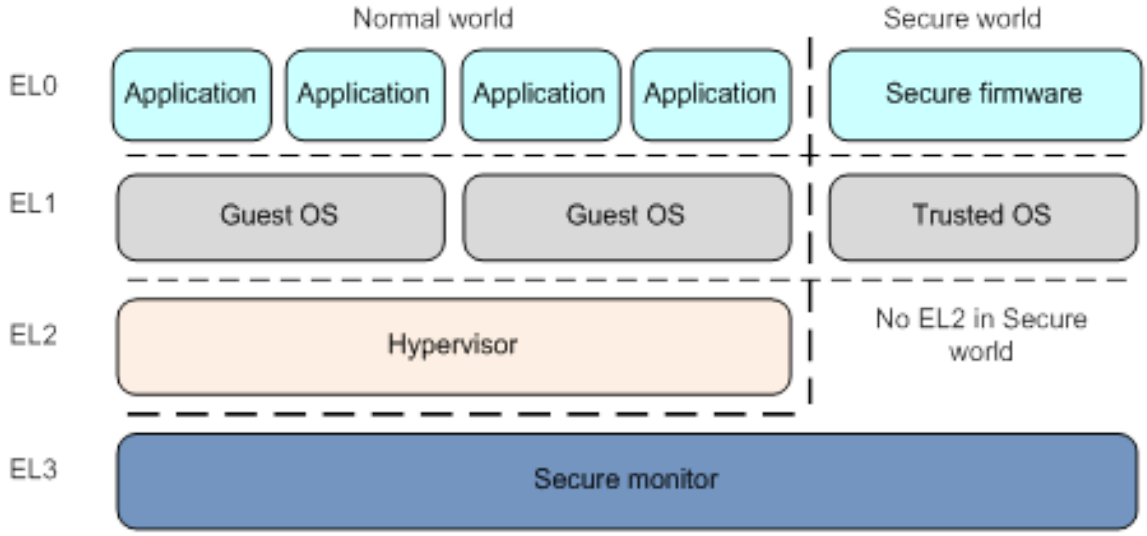

Abbildung 2.1: Tust Zone Exception Levels [11]

Die drei Exception Level die von der TrustZone implementiert werden sind EL1, EL2 und EL3, siehe Darstellung 2.1. Es wird vorgesehen, dass auf dem EL1 Exception Level Betriebssysteme laufen. Die Betriebssysteme können wie Threads von einem Hypervisor umgeschaltet werden. Dementsprechend wurde festgelegt, dass in dem EL2 Exception Level der Hypervisor laufen soll. Das EL3 Exception Level ist für den Secure Monitor vorgesehen. Dieser stellt die höchste Berechtigungsebene dar und ist für die Überwachung und Kontrolle der unteren Ebenen zuständig. User Mode wird auch als EL0 bezeichnet, obwohl es kein Exception Zustand ist. Außerdem startet der Kern immer im höchsten verfügbaren Level.

Da in dieser Arbeit weder Virtualisierung noch Secure Monitor benötigt wird, wurde die TrustZone Erweiterung in Quemo deaktiviert. Damit ist das höchst verfügbare Exception Level in dieser Arbeit EL1.

2.1.2 Exceptions

Wie bereits in Berechtigungen erwähnt, sind Exceptions nötig, um auf eine höhere Berechtigungsebene zu kommen. Dementsprechend wichtig sind diese für die Umsetzung eines Betriebssystems. Exception sind Systemevents, die normalerweise das Eingreifen von privileged Software erfordern, um einen reibungslosen Betrieb des Systems sicherzustellen. Damit ist eine Exception jedes Ereignis, welches eine Unterbrechung des aktuell laufenden Programms verursacht. Wenn eine Exception behandelt wird,

verzweigt die aktuelle Ausführung zu einem Exception Handler anstelle der nächsten Instruktion des Programmes. Nach dem Ausführen des Handlers kehrt die Ausführung zum ursprünglichen Programm zurück. Andere Architekturen bezeichnen den Vorgang unter Umständen als Interrupt. In ARM sind Interrupts extern generierte Exceptions. [12]

Im Unterschied zur x86-Architektur mit 256 Interrupt/Exception Vektoren, besitzt die ARM-Architektur wesentlich weniger Vektoren. Die Anzahl variiert je nach Kern-Design [9]. Die ARMv8 64-Bit Architektur besitzt 4 Sätze mit je 4 Einträgen von Exception Vektoren. Damit sind es 16 Vektoren. Weiterhin hat jedes Exception Level seine eigene Vektor Tabelle mit den jeweils diesen 16 Einträgen. [13]

Welcher Eintrag genutzt wird hängt von folgenden Faktoren hab:

- Der Typ der Exception : Synchronous, Interrupt ReQuest(IRQ), Fast interrupt request(FIQ), System (SError)
- Welcher Stackpointer (SP0 oder SPn) benutzt werden sollte, also damit, ob die Exception vom User Bereich kommt
- Ob die Exception aus einem unterem Exception Level kommt und Aarch64 (64 Bit) oder Aarch32 (32 Bit) benutzt werden soll [13]

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
=+ 0x080	IRQ/vIRQ	
=+ 0x100	FIQ/vFIQ	
=+ 0x180	SError/vSError	
=+ 0x200	Synchronous	Current EL with SPx
=+ 0x280	IRQ/vIRQ	
=+ 0x300	FIQ/vFIQ	
=+ 0x380	SError/vSError	
=+ 0x400	Synchronous	Lower EL using AArch64
=+ 0x480	IRQ/vIRQ	
=+ 0x500	FIQ/vFIQ	
=+ 0x580	SError/vSError	
=+ 0x600	Synchronous	Lower EL using AArch32
=+ 0x680	IRQ/vIRQ	
=+ 0x700	FIQ/vFIQ	
=+ 0x780	SError/vSError	

Tabelle 2.1: Vector Table [13]

Je nach ARM-Architektur kann diese Tabelle variieren und sich auch an festen Adressen befinden. In dem Fall der ARMv8 64-Bit gibt es für jedes Exception Level ein Register, in dem die Base Adresse der jeweiligen Exception Vektor Tabelle gespeichert wird. [13]

Im Unterschied zu x86, wo erst durch auslesen bestimmt werden muss, welche Berechtigungen für einen bestimmten Vektor angewendet werden müssen, werden in der ARM-Architektur die Berechtigungen automatisch durch das Eintreten in die Exception angewendet. [9]

2.1.3 Register

Register sind ein wichtiger Bestandteil eines Prozessors. Sie sind Speicherbereiche für Daten, auf die der Prozessor besonders schnell zugreifen kann und erfüllen unterschiedliche Aufgaben. Die ARM-Architektur besitzt die General Purpose Register und die Special Purpose Register sowie Register, welche für die Konfiguration bestimmter Funktionen des Kernes zuständig sind. Im Folgenden werden die General Purpose Register, Vector Register und die Special Purpose Register erläutert, da diese für Funktionsaufrufe und Exceptions relevant sind.
Die Architektur hat 31 64-Bit General Purpose Register, welche über `X0` bis `X30` und `W0-W30` erreichbar sind. Für den Zugriff auf die vollen 64-Bit wird der Suffix `X` benutzt und für den Zugriff auf 32-Bit wird `W` benutzt. Diese sind in allen Exception Level erreichbar. [14]

Für Funktionsaufrufe werden die General Purpose Register benutzt. Dazu werden diese in vier Gruppen aufgeteilt:

- Parameter and result register `X0-X7`
- Caller-saved temporary registers `X9-X15`
- Callee-saved registers `X19-X29`
- Registers with a special purpose `X8`, `X16-X18`, `X29`, `X30` [15]

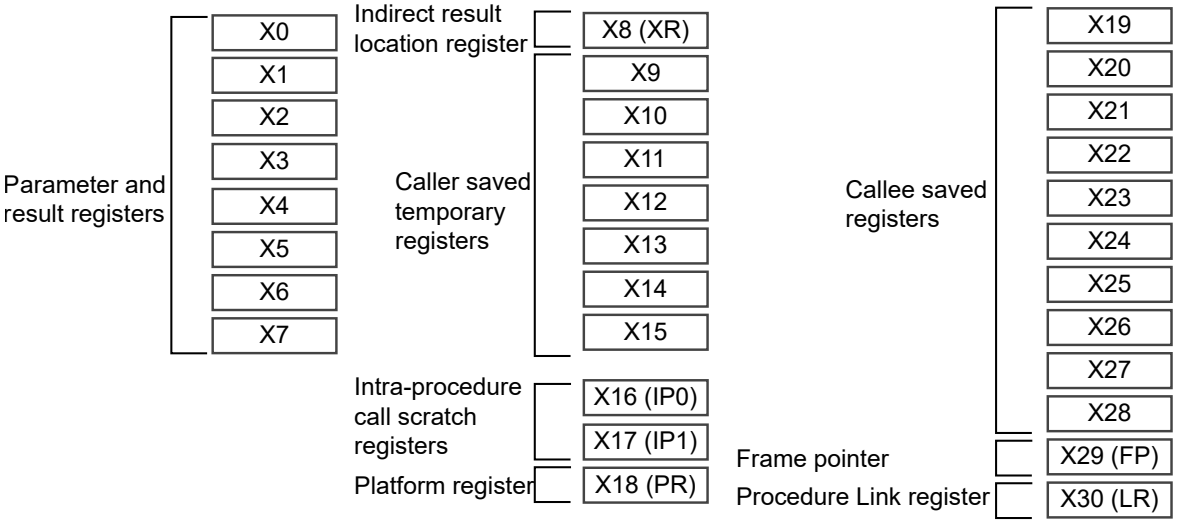

Abbildung 2.2: General Purpose Register [15]

Die "Parameter and result register"werden für das Übergeben von Parametern benutzt. Sollten über 8 Parameter übergeben werden, werden die überschüssigen Parameter auf den Stack gelegt. Außerdemwird `X0` für direkte Return Werte benutzt. [15]

"Caller-saved temporary Register"können von der aufgerufenen Funktion verändert werden, ohne dass diese gespeichert und wiederhergestellt werden müssen. Sollte die aufrufende Funktion diese Werte benötigen, so muss dieser die Werte vorher sichern. [15]

"Callee-saved Register"werden in dem Frame der aufrufenden Funktion gesichert. Sie können in der aufgerufenen Funktion manipuliert werden, solange sie vorher gesichert und danach gespeichert werden. [15]

Die “Registers with a special purpose” werden für unterschiedliche Zwecke genutzt. [15]

`X8` wird für indirekte Return Werte benutzt. In dem Register wird die Adresse auf das indirekte Ergebnis hinterlegt. Dies kann zum Beispiel eine größere C-Struktur sein. [15]

Die Register `X16` (IP0) und `X17` (IP1) sind für intra-procedure-call temporäre Werte vorgesehen. Diese werden von Veneers und Ähnlichem benutzt. Veneers sind kleine Code Snipes, die vom Linker eingefügt werden können. Dies kann zum Beispiel passieren, wenn ein Branch Target außerhalb der Reichweite eines Branch Befehls liegt. Diese Register können von einer Funktion demnach verändert werden. [15]

`X18` ist das Platform Register und wird von Platform ABIs genutzt. [15]

`X29` (FP) ist das Frame Pointer Register [15]

`X30` (LR) ist das Link Register und enthält die Rücksprung Adresse, um zur Aufrufen-Funktion zurückzukehren. [15]

Zu den General Purpose Register kommen 32 128-bit Vektor Register hinzu. Diese werden genutzt, um die Operanten für Floating Point Befehle und Sclare, wie Vektoren für die NEON Befehle, zu speichern. [16] Sie sind nur nutzbar, wenn die Floating Point Unit aktiviert ist. Die Register sind `V0-V31` und wie beim General Purpose Register bestimmt der Suffix, auf wie viel der 128-Bit des Registers zugegriffen wird. Mit `V` werden die vollen 128-Bit adressiert.

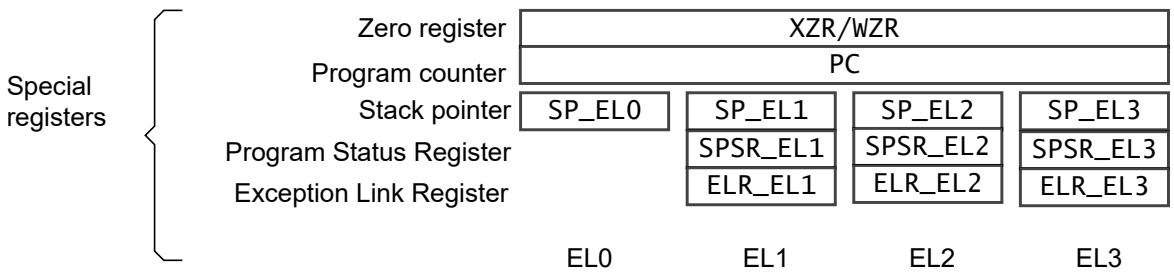

Abbildung 2.3: Special Register [17]

Da jedes Exception Level seinen eigenen Stack Pointer hat, gibt es vier Stack Pointer Register. Diese sind `SP_EL0` bis `SP_EL3`, wobei die Zahl dem entsprechenden Exception Level entspricht, siehe Exceptions. Das Register SP referenziert den aktuell genutzten Stack Pointer und ist von allen Befehlen benutzbar. Wenn sich der Kern in einem anderem Zustand als EL0 befindet, kann der Stack Pointer, der genutzt werden soll, ausgewählt werden. Es kann dabei der Stack Pointer vom aktuellen Exception Level oder der EL0 Stack Pointer ausgewählt werden .

Ebenfalls relevant für Exceptions sind zwei weitere Register. Zum einen das Saved Process Status Register, welches beim Eintritt in eine Exception den Zustand des Kerns speichert und beim Austreten wiederherstellt, [18] und zum anderen das Exception Link Register `ELR`, welches die Rücksprung Adresse für das Verlassen einer Exception beinhaltet. Diese Adresse wird automatisch beim Eintreten in eine Exception geschrieben [19]. Für jedes dieser Register gibt es für jedes Exception Level ab EL1 jeweils eine Version.

2.1.4 Supervisor Call

Wie bereits in Berechtigungen beschrieben ist für den Wechsel von EL0 auf EL1 das Auslösen einer Exception notwendig. Soll nun ein Thread aus dem User Bereich eine Funktion vom Betriebssystem nutzen, muss es möglich sein, eine Exception im User Modus mittels Software auszulösen. Diese Funktion erfüllt der Supervisor Call (SVC).

Der SVC ist eine Software Exception, die durch den Assembler-Befehl `svc <number>` ausgelöst wird. [20]. Beim Ausführen dieses Befehls springt der Kernel in einen Synchronous Exception Typ Vector. Die Konstante hinter dem Befehl wird vom Kern ignoriert und soll von der Software benutzt werden, um zwischen unterschiedlichen SVC-Anfragen zu unterscheiden. Dazu wird die Exception Return Adresse genutzt, um die Instruktion vor dieser Adresse auszulesen. Dann werden die Bits von dem Instruktions Code ausgeblendet, um die Konstante zu extrahieren.

2.1.5 Boot

Der Bootvorgang lädt und initialisiert das Betriebssystem. Der Boot Code muss dazu am Anfang des RAM hinterlegt werden, da der ARM Prozessor mit der Ausführung an dieser Adresse startet.

Unabhängig davon, wie der Prozessor genutzt werden soll, sollten drei Aspekte am Anfang erledigt werden. Als erstes sollte die Floating Point Unit aktiviert werden, auch wenn die Nutzung von Fließkommazahlen nicht vorgesehen ist. Grund dafür ist, dass die Vektor Register sonst nicht genutzt werden können und da der Compiler diese für andere Operationen nutzen kann, kann dies zu Problemen führen. So kann der Compiler zum Beispiel die Vektor Register nutzen, um eine größere Datenstruktur zu kopieren. Außerdem muss der Stack Pointer beim Start gesetzt werden, da dieser beim Start nicht belegt ist. Auf diese Weise steht er für Instruktionen, die den Stack benutzen zur Verfügung. Zuletzt müssen die globalen Objekte initialisiert werden. Dies geschieht durch das Aufrufen der `_init` Funktion. Diese Funktion ist eine Funktion, welche

durch die `.init_array` Section durchgeht und die Konstruktoren aufruft.

Je nach gewünschter Nutzung des Prozessors müssen unterschiedliche Aspekte beim Start durch den Boot Code vorbereitet werden.
Sollte das Nutzen von Exceptions gewünscht sein, muss das Exception Vector Tabellen Register mit der Adresse vom Anfang der Exception Vektor Tabelle gesetzt werden.

Sind Interrupts vom GIC gewünscht, müssen diese vor dem Verlassen vom Boot aktiviert werden. Dazu sollte der Interrupt Masking Bit, mittels des `DAIFClr` PStatefield, entfernt werden.

Ist am Ende des Boot Codes gewünscht, dass Code im User Modus ausgeführt wird, müssen dafür vier Register geladen werden. Zum einen muss Link Return (`X30`) geladen werden, damit eine Rücksprung-Adresse vorhanden ist, falls der Code, der im User Modus ausgeführt werden soll, beendet wird. Ebenfalls sollte der Stackpointer von EL0 gesetzt werden, damit der User Modus einen Stack zum Arbeiten hat. Damit aber auch der EL0 Stackpointer genutzt wird und in EL0 nach dem Verlassen von EL1 gewechselt wird, muss das `SPSR_EL1` Register geladen werden. Es reicht aus, diese Register auf null zu setzen. Zum Schluss sollte das `ELR_EL1` Register gesetzt werden. Dieses soll die Adresse enthalten, zu der nach verlassen von EL1 gesprungen werden soll. Dementsprechend muss die Adresse vom Beginn des User Codes in das Register geladen werden. Mit dem Befehl `eret` kann EL1 verlassen und damit der Bootvorgang abgeschlossen werden.

2.1.6 Generic Interrupt Controller

Der Generic Interrupt Controller (GIC) ist für das Managen der Interrupts zuständig. Er nimmt die Interrupts von Peripherals entgegen, priorisiert diese und leitet sie dann an den zuständigen Kern weiter. [21]

Der GIC wird über Register konfiguriert. Diese Register beeinflussen das Verhalten der einzelnen Interrupt Quellen. Konfigurierbar ist, an welchen Kern der Interrupt weitergeleitet wird, ob der Interrupt aktiviert ist, ob der Interrupt von Software gemasked ist und welche Priorität der Interrupt hat. Der GIC akzeptiert die ausgelösten Interrupts und signalisiert diese dann an jedem verbundenen Kern, bei dem dann entweder eine IRQ- oder FIQ-Exception ausgelöst wird. [21]
Der GIC besitzt aus der Software Perspektive zwei Typen von Interfaces. Diese sind Distributor Interface und CPU Interface.

- Alle Interrupt Quellen sind mit dem Distributor Interface verbunden. Über seine Register werden die Eigenschaften, der einzelnen Interrupts, eingestellt. Die Eigenschaften sind die Priorität, der Zustand, die Sicherheit und ob dieser aktiviert ist. Außerdem kontrolliert er an welchem CPU-Interface und damit an welchen Kern der Interrupt weitergeleitet wird.
- Über das CPU-Interface bekommt ein Kern ein Interrupt signalisiert. Jeder Kern verfügt dabei über ein CPU-Interface. [21]

Die Interrupts werden über eine ID Nummer identifiziert. Diese wird INTID genannt und jede Interrupt Quelle hat ihre eigene INTID. Um zu bestimmen wie der Interrupt zu behandeln ist, kann der IRQ- oder FIQ-Handler diese Nummer aus dem `GICC_IAR` Register lesen. Um dem GIC mitzuteilen, dass der Interrupt behandelt wurde, schreibt der Handler nach dem behandeln des Interrupts, die INTID in das Register `GICC_EOIR` des CPU-Interface. [21] Des Weiteren gibt es die Spurious Interrupt ID 1023 und wird verwendet, um zurückzugeben, dass kein Interrupt aktuell ausstehend ist. [22]

Es gibt vier unterschiedliche Typen von Interrupt Quellen:

- Software Generated Interrupt (SGI) werden von Software ausgelöst und werden meist für inter-core Kommunikation benutzt. Die INTIDs 0 - 15 sind für diesen Typen reserviert.
- Private Peripheral Interrupt (PPI) sind Interrupts, die privat zu einem Kern sind und unabhängig von den Interrupts mit derselben Quelle. Zum Beispiel per Kern-Timer. Die reserviert INTIDs hierfür sind 16-31.
- Shared Peripheral Interrupt (SPI) sind Interrupts, die der GIC zu mehr als einem Kern weiterleiten kann. Die INTIDs 32 - 1020 sind dafür reserviert.
- Locality-specific Peripheral Interrupt (LPI) sind nachrichtenbasierte Interrupt und sind in GICv2 und GICv1 nicht unterstützt. [21]

Des Weiteren können sich die Interrupts in unterschiedlichen Zuständen befinden.

- Inactive: Der Interrupt ist nicht ausgelöst
- Pending: Der interrupt wurde ausgelöst und wartet auf die Bearbeitung vom Kern
- Active: Der Kern behandelt aktuell den Interrupt
- Active and pending: Der Kern behandelt den Interrupt und der GIC hat einen weiteren Interrupt von derselben Quelle Pending [21]

Ein typischer Ablauf dieser Zustände wäre wie folgt [21]:

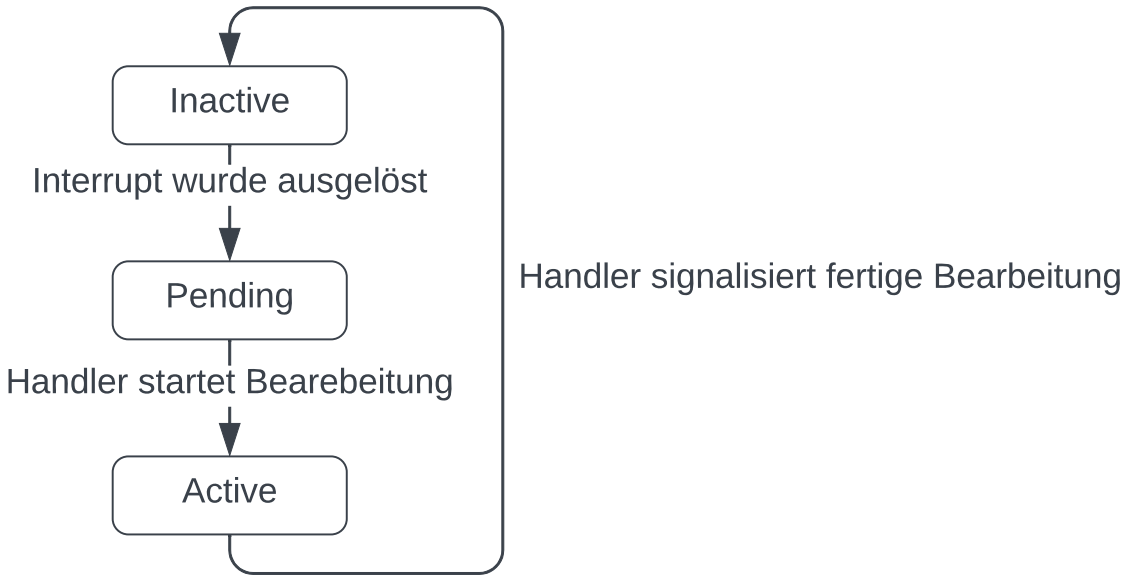

Abbildung 2.4: Zustandsablauf eines Interrupts

2.1.7 Timer

Für das preemptive Umschalten von Threads wird ein Interrupt benötigt, der regelmäßig auslöst. Das bedeutet, ein Timer Interrupt wird benötigt. Die Architektur besitzt dafür den Generic Timer.

Der Generic Timer bietet jedem Kern ein standardisiertes Timer-Framework. Dieser enthält einen System Counter und einen Satz Timers pro Kern. [23, S. 6]

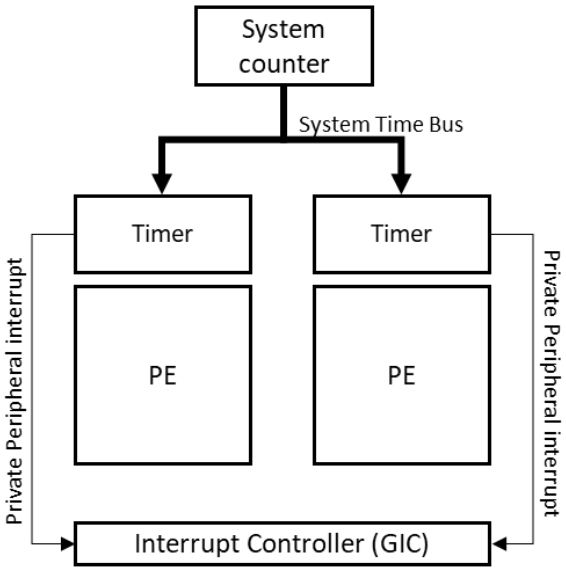

Abbildung 2.5: Timer Framework [23, S. 6]

Der System Counter ist ein Zähler, der mit einer festen Frequenz inkrementiert wird. Der Counter Wert wird an alle Timer von jedem Kern übertragen. Die Timer sind Komparatoren, die einen gesetzten Wert mit dem des System Counters vergleichen. Sie sind so einstellbar, dass sie Interrupts und Events auslösen, wenn bestimmte Punkte in der Zukunft erreicht wurden. Bestimmte Timer gehören zu bestimmen Exception Level. Welche Timer implementiert sind, hängt davon ab, welche Exception Level implementiert sind. [23, S. 6]
Jeder Timer hat die folgenden Register.

Timer	Register prefix	EL<x>
EL1 physical timer	CNTP	EL0
EL1 virtual timer	CNTV	EL0
Non-secure EL2 physical timer	CNTHP	EL2
Non-secure EL2 virtual timer	CNTHV	EL2
EL3 physical timer	CNTPS	EL1
Secure EL2 physical timer	CNTHPS	EL2
Secure EL2 virtual timer	CNTHVS	EL2

Tabelle 2.2: Timer Prefix Tabelle[23, S. 8]

Register	Zweck
<timer>_CTL_EL<x>	Kontroll Register
<timer>_CVAL_EL<x>	Vergebliches Register
<timer>_TVAL_EL<x>	Timer wert

Tabelle 2.3: Timer Register Tabelle[23, S. 8]

Zur Nutzung des Timers muss der gewollte Timer in seinem Controll Register aktiviert werden. Es gibt zwei Möglichkeiten, diesen dann zu konfigurieren, entweder durch das Nutzen des Komparator Register (`CVAL`) oder durch das Nutzen des Timer Registers (`TVAL`).
Wenn das CVAL Register genutzt werden soll, schreibt die Software einen Wert ins Register und sobald der System Counter diesen Wert erreicht oder überschreitet, wird ein Interrupt ausgelöst. [23, S. 9]

```
Timer Condition Met: CVAL <= System Count
```
[23, S. 9]

Wenn das TVAL Register genutzt werden soll, schreibt die Software einen Wert ins TVAL Register. Der Prozessor addiert dann zu diesem Wert den aktuellen System Count und schreibt diesen in CVAL. Der Interrupt wird ausgelöst, sobald der System Count den CVAL Wert erreicht oder überschreitet. [23, S. 9]

```
CVAL = TVAL + System Counter
Timer Condition Met: CVAL <= System Count
```
[23, S. 9]

Auslesen von TVAL wird den verbleibenden Wert bis zum Erreichen des CVAL Wert zurückgeben. Der TVAL Wert wird also dekrementiert, während der System Count inkrementiert. [23, S. 9]

2.1.8 Serielle Schnittstelle

Um mit dem Betriebsystem zu interagieren, wird eine Ausgabe sowie Eingabe benötigt. Eine Möglichkeit dies umzusetzen ist mittels einer seriellen Schnittstelle. Mithilfe der Schnittstelle können Daten zwischen dem Prozessor und einem externen Terminal ausgetauscht werden.

Qemu simuliert für den ARM-Prozessor ein PrimeCell UART (PL011) von ARM. UART steht für Universal Asynchronous Receiver / Transmitter und ist ein serielles Datenübertragungsprotokoll. Dieser kann dementsprechend für eine Ein- und Ausgabe genutzt werden. Der PL011 ist in 2.6 zu sehen.

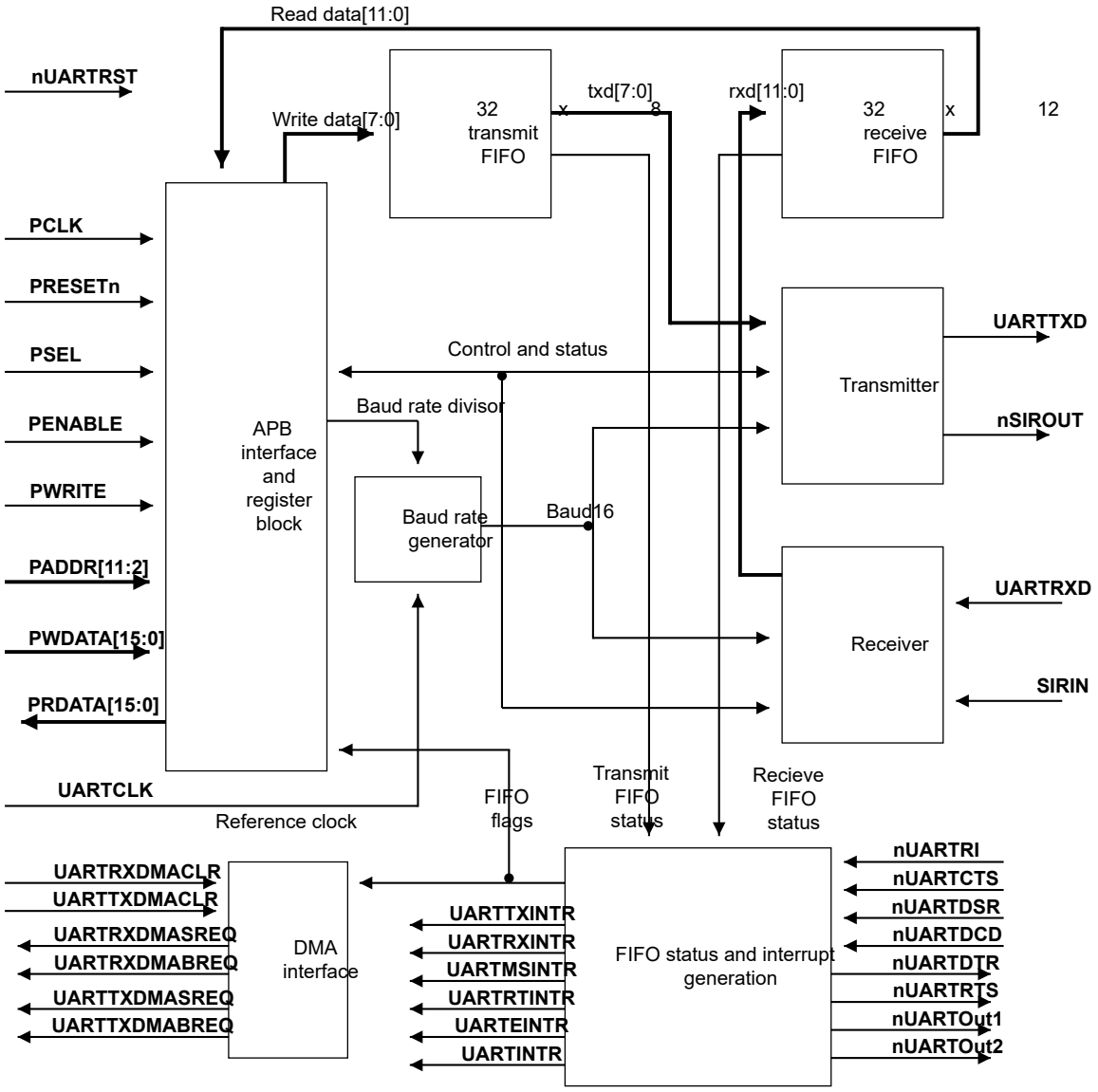

Abbildung 2.6: PL011 [24]

Aus Gründen des Umfangs werden nur die Register beschrieben, die in dieser Arbeit Anwendung finden werden. Diese sind:

- Control Register (UARTC)
- Line Control Register (UARTLCR)
- Flag Register (UAETFR)
- Datenregister (UARTDR)

In dem Control Register kann der UART im Gesamten aktiviert werden und Empfangen und Senden können separat aktiviert und deaktiviert werden. [25]

Im Line Control Register kann eingestellt werden, wie eine UART Übertragung aufgebaut ist und ob die FIFOs vom UART aktiviert sind. [26]

Durch das Flag Register können verschiedene Zustände vom UART abgefragt werden. Für diese Arbeit relevant sind die folgenden Flags

- BUSY zeigt an, ob der UART gerade empfängt oder sendet. BUSY muss überprüft werden bevor an dem Control Register Änderung vorgenommen werden kann.
- TXFF zeigt an, ob der Sende Buffer voll ist. Dieser muss geprüft werden, bevor gesendet werden darf.
- RXFE zeigt an, ob der Empfangs Buffer leer ist. Dieser muss geprüft werden bevor die empfangenen Bytes abgeholt werden dürfen. [27]

Das Daten Register wird genutzt, um Daten zu senden und zu empfangen. Wird ins Daten Register geschrieben, so sendet der UART den Byte durch schreiben in den Buffer. Wird vom Daten Register gelesen, so wird aus dem Empfangs Buffer gelesen. [28]

2.2 Der Kurs

Der Kurs Betriebssystem Entwicklung ist ein Masterkurs der HHU. In der Vorlesung des Kurses sollen laut Modulhandbuch die grundlegenden Konzepte für die Entwicklung eines x86-basierten Betriebssystems vermittelt werden. Diese sind für die Übungen notwendig. Die Inhalte umfassen den Bootvorgang, das x86 Programmiermodell, Interrupts (PIC und APIC), Koroutinen, Threads, Scheduling, Synchronisierung und Treiberarchitektur. Der Schwerpunkt des Kurses soll laut Modulhandbuch in den Übungen liegen. In diesen werden in Einzelarbeit ein x86 basierendes 64-bit Betriebssystem von Grund auf entwickelt. Bei der Programmiersprache für die Übungen hat der Student die Wahl zwischen C++ und Rust. [29, S. 24]

In den Übungen wird dementsprechend ein Projekt bearbeitet, das in mehrere Projektstufen bzw. Aufgaben unterteilt ist. Die Aufgaben bauen aufeinander auf und ergeben am Ende das gesamte Projekt. Diese Aufgaben haben die folgende Gliederung:

- Aufgabe 1: Ein- und Ausgabe
- Aufgabe 2: Speicherverwaltung
- Aufgabe 3: Interrupts
- Aufgabe 4: Nebenläufigkeit
- Aufgabe 5: Timer
- Aufgabe 6: Synchronisierung [30]

Nach Abschluss des Kurses sollen die Studierenden in der Lage sein in C++ oder Rust hardwarenahe Programmierung anzuwenden, Grundlegende Betriebssystemfunktion selbst zu entwickeln und die nebenläufigen Vorgänge (Interrupts und Threads) in einem Betriebssystem selbst zu programmieren und zu synchronisieren. [29, S. 24]

Der Kurs ist laut Modulhandbuch so gestaltet, dass auch Studierende teilnehmen können, die nicht das Bachelor Modul Betriebssysteme und Systemprogrammierung besucht haben. [29, S. 24]

Kapitel 3

Der Kurs in ARM

Im folgenden Abschnitt wird die Implementation des Betriebssystems behandelt. Da die Implementation dem Kurs als Leitfaden folgt, wird jede Aufgabe des Kurses von Anfang bis Ende durchgegangen und im ARM implementiert. Außerdem werden Unterschiede zur x86-Architektur herausgearbeitet und analysiert.

Die Aufgaben aus dem Kurs sind wie folgt:

- Aufgabe 1: Ein- und Ausgabe
  - Aufgabe 1.1: CGA-Bildschirm
  - Aufgabe 1.2: Tastatur
  - Aufgabe 1.3: PC-Lautsprecher
- Aufgabe 2: Speicherverwaltung
  - Aufgabe 2.1: Bump-Allokator
  - Aufgabe 2.2: Listenbasierter Allokator
- Aufgabe 3: Interrupts
  - Aufgabe 3.1: Programmable Interrupt Controller (PIC)
  - Aufgabe 3.2: Weiterleitung von Interrupts an die Geräte-Treiber
  - Aufgabe 3.3: Tastaturabfrage per Interrupt
  - Aufgabe 3.4: Kritische Abschnitte
- Aufgabe 4: Nebenläufigkeit
  - Aufgabe 4.1: Koroutinen
  - Aufgabe 4.2: Warteschlange
  - Aufgabe 4.3: Umbau der Koroutinen auf Threads
  - Aufgabe 4.4: Scheduler
  - Aufgabe 4.5: Multithreading-Anwendung I
- Aufgabe 5: Timer
  - Aufgabe 5.1: Programmable Interval Timer (PIT)

  - Aufgabe 5.2: Umbau des Treibers für den PC-Lautsprecher
  - Aufgabe 5.3 Synchronisierung des Allokators gegenüber Interrupts
  - Aufgabe 5.4 Threadumschaltung mithilfe des PIT
  - Aufgabe 5.5: Testanwendung mit Multithreading
- Aufgabe 6: Synchronisierung
  - Aufgabe 6.1: Synchronisierung mit Interrupt-Sperre
  - Aufgabe 6.2: Semaphore [30]

Wegen begrenzter Zeitressourcenwurde der PC Lautsprecher nicht umgesetzt. Dementsprechend wurden die Aufgaben 1.3 und 5.2 nicht bearbeitet. Der User Mode hat nicht die Berechtigungen Interrupts zu deaktivieren. Folglich ist die Umsetzung einer Interrupt Sperre nicht möglich. Daher wurden die Aufgaben 3.4, 5.3 und 6.1 nicht umgesetzt, weil diese eine Interrupt Sperre nutzen oder umsetzen. Außerdem ist es mit dem UART nicht möglich, einen Interrupt bei jedem Zeichen auszulösen. Wenn der FIFO eingeschaltet ist, ist ein Interrupt nur bei einem eingestellten Füllstand auslösbar. Deshalb wurde die Aufgabe 3.3 nicht bearbeitet.

3.1 Ein und Ausgabe

In der Aufgabe Ein- und Ausgabe sollen die Unteraufgaben Color Graphics Adapter (CGA), Bildschirm und Tastatur bearbeitet werden. Durch die Aufgabe sollen die folgenden Lernziele vermittelt werden.

- Kennenlernen der Entwicklungsumgebung
- Einarbeiten in die Programmiersprache C++
- Hardwarenahe Programmierung: CGA-Bildschirm und Tastatur [31]

3.1.1 CGA Bildschirm

Laut der Unteraufgabe soll eine Ausgabefunktion für Textausgabe und Fehlerausgabe implementiert werden. Diese Ausgabefunktion soll mit einer einfachen Ausgabe von Text geprüft werden. Sie soll ähnlich wie der Cout Stream von der Standardbibliothek Std in C++ funktionieren. Dazu muss, dementsprechend auch ein globales Objekt kout definiert werden. [31]

Von der Lösung des Kurses konnte nicht viel übernommen werden. Die Methoden der Outputstream-Klasse dagegen konnten weitestgehend verwendet werden. Außerdem wurde adaptiert, dass die Outputstream-Klasse einen Stringbuffer erbt und eine `flush` Methode überschrieben werden muss. Dies erfolgt wie in der Kurs Umsetzung durch eine Klasse, die `flush` mit einer Methode überschreibt, welche den Inhalt von dem Buffer auf dem Bildschirm ausgibt.

Aufgrund dessen, dass der CGA nicht vorhanden ist, muss in der `flush` Methode die Ausgabe durch den UART erfolgen.

CGA_Stream.cc C++

```
void CGA_Stream::flush () {
    print (buffer, pos);
    pos = 0;
}
```

stdio.hpp C++

```
namespace peripherals::stdio {
    //...
    class stdout : public lib::basic_ostream<char> {
        //...
        virtual void flush() {
            uart::print(buffer, writePos);
            writePos = 0;
        }
        //...
    }
}
```

Zu erwähnen ist, dass es durch den CGA möglich ist, die Zeichen auf feste Positionen auszugeben. Dies ist durch den UART nicht möglich. Stattdessen werden die Zeichen einzeln in Reihenfolge an den UART übergeben.

CGA.cc C++

```
void CGA::print (char* string, int n, unsigned char attrib) {
    int x, y;
    char* pos;

    getpos(x, y);
    pos = (char*)CGA_START + (y * COLUMNS + x) * 2;

    while(n) {
        switch (*string) {
            case '\n':
                x=0;
                y++;
                pos = (char*)CGA_START + (y * COLUMNS + x) * 2;
                break;
            default:
                *pos = *string;
                *(unsigned char*)(pos + 1) = attrib;
                pos += 2;
                x++;
                if (x >= COLUMNS) {
                    x=0;
                    y++;
                    pos = (char*)CGA_START + (y * COLUMNS + x) * 2;
                }
                break;
        }
        string++;
        if (y >= ROWS) {
            scrollup();
            y--;
            pos = (char*)CGA_START + (y * COLUMNS + x) * 2;
        }
        n--;
    }
    setpos (x, y);
}
```

uart.cpp C++

```
namespace peripherals::uart {
    void putchar(char c) {
        while(flags->TXFF);         // wait until FIFO is not full
        data->DATA = c;
    }

    void print(const char *s, size_t size) {
        for(; size != 0; size--, s++)
            putchar(*s);
        while(!(flags->TXFE));         // wait until transmission is done
    }
}
```

Ein paar Aspekte konnten aus der Kurzlösung übernommen werden, wurden aber anders umgesetzt. Der Stringbuffer wird zu einem Ringbuffer geändert, da diese Klasse später auch für Eingaben genutzt werden sollte. Der Ringbuffer kann für die Ausgabe wie der Buffer aus dem Kurs verwendet werden. Außerdem wird er zu `basic_stringbuf` umbenannt, um näher an der Implementierung vom Stringbuffer aus der Std zu sein. Aus demselben Grund wird auch eine Template Variable hinzugefügt, um zu bestimmen, ob ein char oder ein wchar als Charakter Basis dienen soll. Outputstream wird zu `basic_ostream` umbenannt und mit einem Template versehen, um näher an der Std zu sein. Für das globale Stream Objekt wird mittels `typedef basic_ostream<char> ostream;` ostream definiert und damit das globale Objekt erstellt. Zusätzlich bekommt `basic_ostream` die Möglichkeit, Objekte von der String Klasse zu verarbeiten. Anstelle von `kout` wird `cout` global definiert, damit es ebenfalls näher an der Benennung im Std zu liegen. Da CGA_stream nicht mehr auf CGA basiert, wird der Name auf stdout geändert.

3.1.2 Tastatur

Die Aufgabenstellung gibt vor mit dem Betriebssystem interagieren zu können. Aus diesem Grund sollte ein Tastaturtreiber implementiert werden. Für diese Implementierung sollten keine Interrupts benutzt werden. [31]

Die Implementierung sieht dementsprechend eine Endlosschleife vor, in der die Tastatur abgefragt wird und bei einer Eingabe diese ausgegeben werden soll. Die Aufgabenstellungen schlägt dabei vor, mit der `key_hilt` Methode zu beginnen. Diese Methode soll in einer Schleife darauf warten, dass ein Byte am Datenport liegt. Dazu sollte der `OUTB` Bit im Control Port geprüft werden. Anschließend wird der Byte aus dem Datenport gelesen und in einer Variable abgelegt. Infolgedessen muss mittels des `AUXB` Bit im Control Register geprüft werden, ob der Byte von der Tastatur und nicht der Maus stammt. Danach kann mittels `key_decode` der Byte übersetzt werden. Falls die Übersetzung erfolgreich war, gibt `key_decode` true zurück und das Ergebnis in `gather` kann zurückgegeben werden. Falls die Übersetzung fehlschlug, sollte `invalid` zurückgegeben werden. Wie bereits erwähnt, soll die Tastatur in einer Endlosschleife abgefragt werden, dementsprechend soll `key_hilt` in einer Endlosschleife aufgerufen werden. [31]

Da keine PS2 Tastatur in der Qemu ARM Simulation vorhanden ist, konnte von der Umsetzung aus dem Kurs nichts verwendet werden. Stattdessen werden die Tastatureingaben über UART umgesetzt.

uart.cpp C++

```
namespace peripherals::uart {
    char getChar() {
        while(flags->RXFE);      // wait until there is something to read
        char c = data->DATA;
        putchar(c);
        return c;
    }
}
```

Um ein Zeichen einzulesen, muss in einer Schleife auf das Einkommen eines Zeichens gewartet werden. Dies kann mittels des `RXFE` Bit in dem Flags Register geprüft werden. Anschließend kann durch das Auslesen vom Datenregister das Zeichen direkt ausgelesen und zurückgegeben werden. Die Zeichen müssen weder dekodiert, noch geprüft werden.

Zusätzlich wird ein Eingabe-Stream im Stil von Std `cin` umgesetzt. Dafür wird ein `basic_inputstream` umgesetzt, der den Stringbuffer nutzt. In der Stdin Klasse wird dann anstelle `flush` `readline` überschrieben. `readline` soll dann mittels UART die Zeichen abholen, bis mittels der Eingabetaste die Eingabe bestätigt wird. Analog zu cout wird ein globales cin Objekt definiert.

3.2 Speicherverwaltung

In der Aufgabe Speicherverwaltung soll sowohl ein Bump-Allocator als auch ein Listbasierter Allocator implementiert werden. Damit soll das folgenden Lernziel erreicht werden:

- Verstehen wie eine Speicherverwaltung funktioniert und implementiert wird [32]

3.2.1 Bump-Allocator

Laut Aufgabenstellungen des Kurses soll ein Bump-Allocator umgesetzt werden. Dem Allocator ist nur Heap Anfang und Ende bekannt. Mit einer Variable `next` wird die aktuelle Adresse im Heap vermerkt, ab der noch Speicher zur Verfügung steht. Bei einer Allocation wird `next` um die angefragten Bytes weiter gesetzt, solange das Heap Ende nicht erreicht ist. Bei einer Speicherfreigabe soll nichts geschehen. Außerdem soll der Heap so früh wie möglich beim Start des Betriebssystems initialisiert werden. `new` und `delete` sind bereits überschrieben, um die entsprechenden Funktionen des Allocators aufzurufen. Dazu gibt es ein globales Bump Allocator Objekt. Außerdem wird `_ZdlPv` in Boot auskommentiert, da diese von `delete` nicht mehr angesprungen werden muss. [32]

Bei der Umsetzung in ARM konnte die Umsetzung des Bump Allocators vollständig übernommen werden. Jedoch werden ein paar Änderungen vorgenommen, die nicht zwingend notwendig sind. Zum einen wird der Speicherbereich für den Heap mittels des Linkers reserviert, anstelle Speicher im Code zu reservieren. Auf diese Weise ist dieser von Anfang an vorhanden und muss nicht zuerst im Code definiert werden. Die

Adressen werden als Linker Symbole dem Allocator Code übergeben. Des Weiteren wird darauf verzichtet, den Allocator als Klasse umzusetzen. Stattdessen wird er als Namespace umgesetzt. Dies hat mehrere Gründe. Zum einen verhindert die Nutzung eines Namespaces die Definition mehrerer Allocator Objekte. Esbesteht keine Notwendigkeit mehr, ein globales Allocator Objekt erstellen zu müssen. Der zu verwendende Allocator kann mittels eines Namespace alias gesetzt werden.

allocator.hpp C++

```
1 namespace allocator = Allocator::BumpAllocator;
```

Die Nutzung eines Namespace macht auch auf den ersten Blick deutlich, dass es sich um statische Funktionen handelt und nicht um Methoden eines Singleton Objekts. Dies erhöht die Lesbarkeit des Codes und verbessert die Performance. Beim Aufruf einer Funktion muss kein Objekt Zeiger mehr übergeben werden, welcher für den Zugriff auf die Objektvariablen genutzt werden würde. Stattdessen kann nun, mittels direkter Adressen, direkt auf die Variablen zugegriffen werden. Dies sollte dem Compiler erlauben, die Funktion mit weniger Instruktionen umzusetzen.

3.2.2 Listenbasierter Allocator

Als nächstes soll, laut Aufgabenstellungen, ein Allocator implementiert werden, welcher auch Speicher freigeben kann. Dazu soll ein List-basierter Allocator umgesetzt werden.

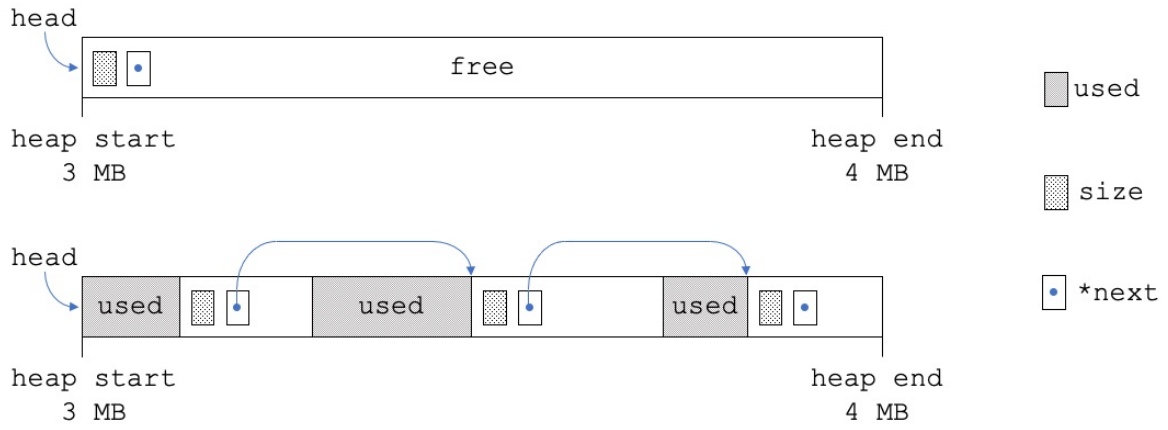

Abbildung 3.1: Listenbasierter Allocator [32]

Nach der Initialisierung gibt es einen großen freien Speicherblock, welcher als einziger Eintrag in der Freispeicherliste steht. Bei der Allokation wird die Freispeicherliste nach einem passenden Block durchsucht und der erste Block mit passender Größe gewählt. Sollte der Rest vom Block groß genug sein, die Metadaten für einen Listeneintrag zu speichern, wird dieser Rest in die Freispeicherliste eingetragen. Bei einer Freigabe soll der freizugebende Block in der Freispeicherliste wieder eingehängt werden. Optional können benachbarte Blöcke in der Liste verschmolzen werden. Dafür muss in der Liste gesucht werden. Zu Bedenken ist, dass bei einer Allokation 8 Byte zusätzlich reserviert werden müssen, um die Längen Informationen des Speicherblocks zu speichern. Diese sind bei der Freigabe notwendig. Die Länge kann an den Anfang des Speicherblocks

abgelegt werden und die Adresse nach der Länge Information wird dann von der Funktion zurückgegeben. [32]

Für die Umsetzung des listenbasierten Allocators in ARM gelten dieselben Änderungen wie für den Bump-Allocator. Welche Art von Allocator umgesetzt wird und ob für x86 oder ARM, spielt für die Umsetzung keine Rolle.

3.3 Interrupts

In der Aufgabe Interrupts soll der Programmable Interrupt Controller implementiert werden, Weiterleitungen von Interrupts an Gerätetreiber sowie die Behandlung von kritischen Abschnitten. Damit sollen die folgenden Lernziele vermittelt werden:

- Funktionsweise des Interrupt-Controllers verstehen
- Behandlung von Interrupts implementieren, am Beispiel der Tastatur
- Kritische Abschnitte (Synchronisierung) verstehen und einsetzen [33]

3.3.1 Programmable Interrupt Controller (PIC)

Laut Aufgabenstellungen soll zuerst die Interrupt Verarbeitung aktiviert werden. Diese soll durch die Umsetzung des Tastatur Interrupt geprüft werden. Dazu müssen zuerst die leeren Methoden der PIC Klasse implementiert werden. Die Funktion `int_disp` soll zunächst eine einfache Textausgabe mit der Vektor Nummer des ausgelöst interrupt ausführen. Für die Implementierung des Tastatur Interrupt soll `plugin` in `keyboard.cc` umgesetzt werden. Diese Funktion soll `pic.allow` aufrufen, um den Interrupt zu aktivieren. In der Main Funktion soll, mit der `kb.plugin` Methode, der Tastatur Interrupt aktiviert werden. Des Weiteren muss mit `cpu.enable_int()` Interrupts zugelassen werden und mit `init_interrupts()` muss PIC initialisiert werden. [33]

Nun soll, wenn das System läuft und eine Taste gedrückt wird, eine Textausgabe erscheinen. Dies funktioniert nur, bis der Buffer der Tastatur voll läuft, da die Daten der Tastatur nicht abgeholt werden. Und damit stoppt das Auslösen von Interrupts. Die Interrupt Verarbeitung funktioniert nur korrekt, solange der Kern läuft, demzufolge soll `main` nicht beendet werden. [33]

Bei der Umsetzung in ARM kann die Interrupt-Verarbeitung nicht mit dem Tastatur Interrupt benutzt werden. Dies liegt daran, dass UART für die Tastatureingabe genutzt wird und UART nicht die Möglichkeit hat, ein Interrupt für jedes Zeichen auszulösen, wenn der Buffer vom UART aktiviert ist. Dennoch wird eine Interrupt Verarbeitung für den Timer benötigt.

Da ARM kein PIC besitzt, wird von der Umsetzung in x86 nichts genutzt. Statt des PIC besitzt ARM einen Generic Interrupt Controller (GIC). Der GIC löst entweder

einen IRQ Exception oder eine FIQ Exception aus, abhängig davon, wie die Interrupt Quelle in GIC konfiguriert wurde. Der Unterschied zwischen den Exception Types ist, dass FIQ eine höhere Priorität hat als IRQ. In der Implementierung dieser Arbeit reicht die Implementierung des IRQ mit der Weiterleitung aller genutzten Interrupt Quellen an diese aus.

Der Interrupt Vektor für den IRQ wird durch das Laden der Vektor Tabelle in Boot eingestellt. Am Anfang dieses Vektors müssen alle General Purpose und Vektor Register auf dem Stack gesichert werden.

boot.asm

```
IRQCall:
    str X0, [SP, #-8]!
    str X2, X1, [SP, #-16]!
    // and so forth
    str X28, X27, [SP, #-16]!
    str X29, [SP, #-8]!

    str Q1, Q0, [SP, #-32]!
    // and so forth
    str Q31, Q30, [SP, #-32]!

    //...
    bl irg_handler
    //...

    ldr Q31, Q30, [SP], #32
    // and so forth
    ldr Q1, Q0, [SP], #32

    ldr X29, [SP], #8
    ldr X28, X27, [SP], #16
    // and so forth
    ldr X2, X1, [SP], #16
    ldr X0, [SP], #8

    ret

curr_el_spx_IRQ:
    //...

    str x30, [SP, #-8]!
    bl IRQCall
    ldr X30, [SP], #8

    //...
    eret
```

Grund dafür ist, dass beim Aufrufen der C Funktion, die der IRQ Händler sein soll, diese Register korrumpiert werden könnten. Die C Funktion `irg_handler` entspricht dann der `int_disp` Funktion, mit dem Unterschied, dass keine Vector Nummer als Parameter übergeben wird. Der GIC besitzt keine Vektor Nummer, um die interrupt Quelle zu identifizieren. Stattdessen wird eine interrupt id genutzt (INTID). Die ID des aktuellen Interrupt kann dem `GICC_IAR` Register entnommen werden. Des Weiteren muss am Ende vom Interrupt Handler die INTID in `GICC_EOIR` geschrieben werden, um den Interrupt Status im GIC zurückzusetzen.

Anders umgesetzt wird ebenfalls, zu welchem Zeitpunkt die Interrupts aktiviert werden. Da auf eine `main` am Ende verzichtet werden soll, werden die Interrupts bereits

in Boot aktiviert. Aus demselben Grund wird GIC in dem SVC Call, der für dem Betriebssystem Start genutzt wird, initialisiert. Dieser SVC-Call wird in Boot durchgeführt.

3.3.2 Weiterleitung von Interrupts an die Geräte-Treiber

Im nächsten Schritt soll eine Interrupt Weiterleitung implementiert werden, damit die Interrupts an Treiber weitergeleitet werden. Dazu soll ein Treiber eine Interrupt-Service-Routine(ISR) implementieren und registrieren. ISR ist eine Schnittstelle, die nur aus einer `trigger` Funktion besteht, die vom Treiber überschrieben werden soll. Zu beachten ist, dass der Interrupt-Dispatcher mit der Vektor Nummer arbeitet und nicht mit der IRQ Nummer vom PIC. Zur Verwaltung der Treiber dient die `IntDispatcher` Klasse. Diese speichert alle registrierten Treiber in einem ISR Array. Mittels `assign` wird ein Treiber beim `IntDispatcher` registriert und in dem Array unter seiner Vektor Nummer eingetragen. Bei einem Interrupt soll mittels der `report` Funktion die Trigger Funktion ausgeführt werden, von dem Treiber Objekt korrespondieren mit der Vektor Nummer. Falls kein Treiber registriert wurde, soll eine Fehlermeldung ausgeben werden und das System gestoppt werden. Plugin soll nun so verändert werden, dass das Keyboard sich selbst mit assign beim `IntDispatcher` registriert. `trigger` vom Keyboard soll nun eine einfache Textausgabe durchführen. [33]

Bei der Umsetzung in dieser Arbeit können folgende Dinge nicht realisiert werden. Da kein Tastatur Interrupt umgesetzt wird, muss auch kein `plugin` modifiziert werden. Anstelle der Keyboard Trigger Funktion wird die `trigger` Funktion vom Timer genutzt, damit eine Ausgabe beim Triggern eines Interrupts vorhanden ist.

Da der Timer interrupt ein fester Bestandteil des Betriebssystems ist, besteht keine Notwendigkeit ihn dynamisch bei einem `IntDispatcher` zu registrieren. Stattdessen wird mittels eines Switch Case die `trigger` Funktion vom Timer direkt im Handler aufgerufen. Der Switch Case entscheidet anhand der INTID, welche Funktion aufgerufen werden soll. Damit besteht keine Notwendigkeit, eine `IntDispatcher` oder `ISR` Klasse umzusetzen. Damit muss der Timer keine Klasse sein, die `ISR` vererbt. Stattdessen kann das Timer Objekt ein Namespace sein. Dies ist zu bevorzugen, da nur ein Timer Objekt existieren soll. Mittels eines Namespace gleicht die Schreibweise dem eines statischen Objekts und ein globales Objekt für den Timer ist nicht mehr notwendig. Sollte trotzdem eine dynamische Registrierung von Treiber gewünscht sein, so kann dies, anstelle von einer `ISR` Vererbung, mittels eines Funktionszeigers realisiert werden. Damit kann bei Singleton Objekten diese immer noch mit Namespaces umgesetzt werden. Für die Lesbarkeit kann für den Funktionszeiger ein Typ mittels `typedef` definiert werden. Dieser würde, anstelle von `ISR`, bei der Erstellung von Array im IntDispatcher genutzt werden.

3.4 Koroutinen und Threads

In der Aufgabe Koroutinen und Threads sollten zuerst Koroutinen realisiert werden. Dann eine Warteschlange in Vorbereitung auf den Scheduler. Daraufhin werden die Koroutinen zu Threads geändert und danach als weitere Unteraufgabe der Scheduler

umgesetzt. Zuletzt wurde zum Testen die erste Multithreading Anwendung durchgeführt.
Durch die Aufgabe sollen die folgenden Lernziele erreicht werden.

- Auffrischen der Assembler Kenntnisse
- Verständnis der Abläufe bei einem Koroutinen-Wechsel
- Unterschied zwischen Threads und Koroutinen
- Verstehen wie ein Scheduler funktioniert [34]

3.4.1 Koroutinen

In der Teilaufgabe Koroutinen sollen Koroutinen und das Umschalten zwischen diesen implementiert werden. Koroutinen sind eine Vorstufe zu Threads. Sie unterscheiden sich zu Threads in der Weise, dass Sie die Kontrolle freiwillig an das Betriebssystem abgeben und einen Wechsel verursachen. Das bedeutet, dass kein präemptives Umschalten von Routinen passiert. Um das Umschalten umzusetzen, sollten zuerst die Assembler Funktionen `_coroutine_start` und `_coroutine_switch` implementiert werden. `_coroutine_switch` sichert den Zustand der aktuellen Routine auf dem Stack der Routine. Dazu werden alle Register, die wichtig für den Zustand sind, auf den Stack der Routine gesichert. Danach soll die Funktion von der nächsten Routine den Zustand wieder laden. Also die Registerwerte von dem Stack der Routine laden. Der Stackpointer für diese Operationen soll in einer Koroutinen Klasse hinterlegt sein. Die `_coroutine_switch` Funktion bekommt die zwei Objekte aktuelle Routine und nächste Routine von dieser Klasse als Parameter übergeben. Die Funktion `_coroutine_start` soll den Stack eines Koroutinen Objekts initialisieren, indem sie den Stack mit initial Register Werten belädt und auf diese Weise für das erste Laden durch `_coroutine_switch` vorbereitet. Somitsollen die Koroutinen Objekte mittels eines Zeiger verkettetet werden. Diese soll dazu dienen, eine einfache Ablauf-Reihenfolge zu haben. [34]

Zum Testen sollen drei Koroutinen erstellt werden, die jeweils ihren eigenen Zähler hochzählen und diese Zähler auf einer festen Position auf dem Bildschirm ausgegeben werden. Die drei Routinen sollen an unterschiedlichen Stellen auf dem Bildschirm ausgegeben werden. Dazu soll die `Run` Funktion der Objekte überschrieben werden. Dieser Test zeigt dann durch die Zähler, dass die Routinen unabhängig voneinander laufen können. [34]

In der ARM-Umsetzung können und werden folgende Aspekte equivalent umgesetzt. Der Zustand der Routinen wird wie in x86 auf eine Stack abgelegt und wiederhergestellt. Der Stackpointer wird ebenfalls auf einem Objekt gesichert.

Nicht equivalent umgesetzt werde folgende Aspekte. Die Implementierung von `_coroutine_start` und `_coroutine_switch` da sich die Register in ARM zu den in x86 unterscheiden, siehe Kapitel Register. Infolgedessen müssen die meisten Register bereits am Anfang eines Interrupts auf einen Stack gesichert und am Ende des Interrupts wiederhergestellt werden. Diese Register sind alle General Purpose Register,

alle Vector Register und das SPSR Register, welches das PState enthält und damit den Zustand des Kerns selbst (z.B. die Flags der ALU). Dementsprechend muss in der Interrupt Routine nur der Stack des aktuellen Threads als genutzter Stack ausgewählt werden. Dies kann durch das Schreiben in das `SPsel` Register erreicht werden. Durch das `SPsel` Feld wird ausgewählt, ob der Stack Pointer von EL0 oder der des aktuellen Exception Level genutzt werden soll. Dieses Feld befindet sich im `PState` Register und ist nur über sein eigenes Special Purpose Register erreichbar. Mit `MSR SPSel, #0b1` wird der Stackpointer vom aktuellem Exception Level ausgewählt. Mit `MSR SPSel, #0b0` wird der Stackpointer von EL0 ausgewählt.

Coroutine.asm asm

```
Coroutine_switch:
    mov         eax,[4+esp]             ; regs_now
    mov         [ebx_offset+eax],ebx    ; nicht-fluechtige Register speichern
    mov         [esi_offset+eax],esi
    mov         [edi_offset+eax],edi
    mov         [esp_offset+eax],esp
    mov         [ebp_offset+eax],ebp

    mov         eax,[8+esp]             ; regs_then
    mov         ebx,[ebx_offset+eax]    ; Register wieder herstellen
    mov         esi,[esi_offset+eax]
    mov         edi,[edi_offset+eax]
    mov         esp,[esp_offset+eax]
    mov         ebp,[ebp_offset+eax]

    ret                                  ; Coroutinenwechsel !
```

Nachdem alle General Purpose Register auf dem Stack gesichert werden, wenn `_coroutinen_switch` erreicht wurde, muss abschließend das Exception Return Register gesichert, der Stack Pointer für EL0 aktualisiert und danach das Exception Return Register wiederhergestellt werden. Das Exception Return Register enthält die Rücksprung Adresse, zu der nach Beenden der Exception gesprungen wird . Da dies ein extra Register ist, wird dieses durch das Benutzen von Sprüngen mit dem Linker Register nicht korrumpiert.

Eine weiterer Unterschied ist, dass die `_coroutine_switch` Funktion nur die nächste Routine als Parameter übergeben muss. Da der Zugriff auf das aktuelle Routinen Objekt über den Zeiger in dem `TPIDR_EL1` Register erfolgen kann. Das `TPIDR` Register besitzt den Zweck entweder die ID des aktuellen Threads zu enthalten oder den Zeiger auf das Thread Objekt. Dieses Register wird vom Kern selber nicht genutzt und soll vom Betriebssystem verwaltet werden. Dementsprechend wird dieses Register bei einem Wechsel der Routine aktualisiert.

dispatcher.hpp C++

```
namespace kernel::dispatcher {
    inline void contextSwitch(Thread* next) {
        auto currentProcess = (Thread*)registers::TPIDR_EL0::get();

        // get stack pointer and use x28
        __asm __volatile__ ("mrs x28, SP_EL0");
        // save exception return address
        __asm __volatile__ ("mrs x0, ELR_EL1");
        __asm __volatile__ ("str X0, [x28, #-8]!");

        // save stack pointer to current pcb
        __asm __volatile__ ("mov %[sp],x28": [sp]"=r"
          (currentProcess->stackPointer));
        registers::TPIDR_EL0::set((int64_t)next);
        // load stack pointer from next pcb
        __asm __volatile__ ("MOV x28,%[sp]": : [sp] "r" (next->stackPointer));

        // load exception return address
        __asm __volatile__ ("ldr X0, [x28], #8");
        __asm __volatile__ ("msr ELR_EL1, x0");
        // set stack pointer
        __asm __volatile__ ("msr SP_EL0, x28");
    }
}
```

Die Funktion `_coroutinen_start` wird in der Form nicht umgesetzt. Grund dafür ist, dass für das Starten von Koroutinen derselbe Code ausgeführt werden müsste, wie die `_coroutinen_switch` Funktion bereits ausführt. Damit die nötigen Werte auf dem Stack für das Verlassen des Interrupts vorhanden sind, wird beim Erstellen des Routinen Objekts der Stack mit den nötigen Werten vorgeladen. Nun wird in diesem Schritt ebenfalls der Exception Return Wert auf dem Stack gelegt. Dieser entspricht der Code-Adresse vom Anfang des Routinen-Codes.

Zum Testen werden nur zwei Routinen benutzt, welche einen Zähler haben und den Zählerwert auf der Ausgabe ausgeben sollen. Es ist nicht möglich, Zähler auf festen Position auszugeben, weil kein CGA vorhanden ist und UART als Ausgabe benutzt wird. Ersatzweise wird bei jedem Umschalten eine neue Zeile auf der Ausgabe ausgegeben, um zu prüfen, ob sich die Routinen korrekt abwechseln. Mittels Carriage Return kann der Zähler auf eine Zeile fest hochgezählt werden. Um außerdem zu überprüfen, ob die richtigen Werte nach einem Umschalten in den Registern liegen, werden diese manuell mit dem Debugger überprüft. Dazu werden die Register auf den Stack mit den Zahlen eins aufwärts initialisiert und dann kontrolliert, ob die Register diese Zahlen nach dem Umschalten enthalten.

Ein paar Aspekte hätten equivalent umgesetzt werden können, werden aber bewusst anders realisiert. Zum einen werden direkt Threads umgesetzt und auf den Zwischenschritt mit Koroutinen wird verzichtet. Dies hat zwei Gründe. Der erste Grund ist, dass ein Wechsel von Routinen nur auf dem Exception Level geschehen kann, da ein paar der Register nur im Exception Zustand zugreifbar sind. Der zweite Grund ist, dass die meisten Register durch den Interrupt gesichert werden. Dies bedeutet, dass eine Implementation von nicht präemptiven Umschalten sich nicht wesentlich zur Implemention von präemptive Umschaltenum unterscheidet. Damit bot sich an direkt präemptives Umschalten zu implementieren. Ebenfalls werden die Thread Objekt nicht direkt verkettet. Stattdessen wird später eine generische Liste umgesetzt, die als generische War-

teschlange dienen soll. Der Grund dafür ist, dass dies intuitiver sein sollte und beim Testen die Warteschlange separat vorab getestet werden kann. Außerdem sollten beim Testen des Umschaltens zwei Threads ausreichend sein, welche abwechselnd umgeschaltet werden. Die letzte Änderung, die durchgeführt wird, ist, dass sich der Code eines Threads nicht in der Thread-Klasse als `run` Methode befindet. Anstelle dessen werden separate C-Dateien erstellt, die separat übersetzt werden und dann vom Linker in ihre eigene Speicherbereich gelegt werden. Diese Änderung wurde durchgeführt, um den Code, der von einem Thread ausgeführt werden soll, weiter vom Betriebssystem zu trennen. Dadurch ist dieser näher an einem Programm, das separat erstellt und geladen wurde. Des Weiteren macht dies eine Umstellung von Threads auf Prozessen einfacher, sollte dies in der Zukunft gewünscht sein.

scipt.ld `ld`

```
/* ... */
_idle_start = .;
.boot.idle : { KEEP(*(.boot.idle))}
.idle :  { liblib.a:idle.cpp.obj }
. = ALIGN(8);
. += 0x1000; /* 4kB of stack memory */
_idle_stack_top = .;
/* ... */
```

Eine wichtige Bemerkung hier ist, dass die Reihenfolge der Sektion wichtig ist. Die Sektion des Betriebssystems muss am Ende des Scripts sein, da die Sektion den gesamten Code enthalten soll, welcher nicht schon anderen Sektionen zugewiesen wurde. Wenn die Sektion am Anfang wäre, würde sie auch den Code der Threads enthalten und die Thread Sektionen würden leer sein.

Der Code kann mit dem Laden durch den Linker nicht mit virtualisierten Speicheradressen angesprochen werden. Die Adressen werden vom Linker beim letzten Linken auf die physikalischen Adressen angepasst. Dadurch sind die Adressen nicht in Relation zur Adresse Null, sondern in Relation zu ihrem Speicherbereich im gesamten Speicher. Dies macht die Nutzung von Paging zur Virtualisierung des Threads nicht möglich und damit kann der Thread nicht als Prozess umgesetzt werden. Sollte eine Umsetzung von Prozessen gewünscht sein, so müssten diese dynamisch geladen werden.

3.4.2 Warteschlange

Der Scheduler arbeitet mit einer Warteschlange. Laut der Aufgabenstellung, soll diese eine einfach verkettete Liste sein, bei der am Ende immer eingefügt wird und am Anfang immer entfernt wird. Das Entfernen kann bei einem Thread Switch geschehen und das Einfügen beim Start eines Threads. [34]

Da die Threads in dieser Arbeit nicht direkt verkettet werden, siehe Kaptiel Koroutinen, kann die Implementierung von dem Kurs für die Warteschlange nicht übernommen werden. Stattdessen wird eine generische Warteschlangen Klasse umgesetzt. Dabei wird versucht, sich an der Benennung von der Queue Klasse aus C++ Std zu orientieren. Die Queue Klasse aus Std implementiert eine Warteschlange mittels eines Containers von einem generischen List-Types. Dies wird in dieser Arbeit ebenso ausgeführt. Dies

hat den Vorteil, dass eine Liste Klasse für das Speichern der Thread Objekte selbst verfügbar ist. Dementsprechend wird zuerst eine generische Liste implementiert und dann in einer Queue Klasse als Container genutzt, auf dem die Methoden der Queue Klasse angewendet werden. Die Liste wird als einfach verkettete Liste mit head und tail implementiert. Auch hier wurde möglichst versucht, sich an der List Klasse aus Std zu orientieren. Gleichermaßen wird ein Iterator für die Liste implementiert, damit auf die Liste eine einfache for each Schleife anwendbar ist.

Nun existiert ein privates `list` Objekt mit dem Typ `Thread` im Kernel Namespace, um die Threads Objekte zu speichern. Eine private Variable in einem Namespace kann durch das wrappen dieser in einem unbenannten Namespace realisiert werden. Damit kann nur innerhalb des Kernels auf diese Variable zugegriffen werden. Im Gegensatz dazu wird die für die Scheduler Warteschlange als `queue` Objekt vom Typ eines Threads Zeiger umgesetzt.

3.4.3 Umbau der Koroutinen auf Threads

Nun sollen die Koroutinen in Threads umgebaut werden. Dazu soll die `Coroutinen` Klasse zu "Thread" umbenannt werden. Namen der Klasse, Konstruktoren und Methoden sollen angepasst werden. Die Methode `SetNext` soll entfernt werden und `Switch2next` wird zu `switchTo` unbenannt. Der Threads Konstruktor soll nun eine eindeutige Thread-ID bekommen. Diese soll mit einer globalen Variable, die inkrementiert wird, umgesetzt werden. Der Stack soll nun im Konstruktor dynamisch angelegt und im Destruktor gelöscht werden. [34]

Da die Implementierung von Koroutinen übersprungen wurde und direkt Threads implementiert wurden, existiert keine Notwendigkeit für einen Umbau. Es werden jedoch zwei Änderungen durchgeführt. Zum einen werden die Thread-IDs nicht durch das Inkrementieren einer globalen Variable erzeugt. Stattdessen wird die Liste der Threads durchgegangen und dabei wird die nächste unbenutzte Zahl, von 1 an, gesucht. Zum anderen wird der Stack Speicher nicht im Konstruktor dynamisch angelegt. Stattdessen wird der Stack Speicher mit dem Linker reserviert. Damit befindet sich der Stack Speicher in derselben Sektion wie der Thread Code anstelle auf dem Betriebssystem Heap.

3.4.4 Scheduler

Laut Aufgabenstellung folgt nun die Implementierung des Schedulers. Dieser verwaltet die Threads in einer Bereit-Warteschlange und gibt die Kontrolle über die CPU reihum den Threads in der Warteschlange. Es wird keine Priorität umgesetzt und berücksichtigt. Außerdem sollte der aktuell laufende Thread nicht in der Warteschlange gespeichert werden. Falls die Warteschlange leer ist, soll der Idle Thread laufen. Dieser soll bei der Initialisierung des Schedulers erstellt werden. Gemäß der Aufgabenstellung wird die `Dispatcher` Klasse vorgegeben, die den aktuell laufenden Thread verwalten soll. Für das Umschalten der Threads soll die `SwitchTo` Methode der `Dispatcher` Klasse benutzt werden. Die Klasse enthält auch die Methode `start`, um einen Thread anzustoßen. Bei einem Threadwechsel sollte der Kopf der Warteschlange entfernt werden und durch `life` in dispatcher.cc referiert werden. Gibt ein Thread die Kontrolle

durch `yield` ab, wird dieser am Ende der Warteschlange eingefügt. [34]

Bei der Umsetzung in ARM konnte die Umsetzung des Schedulers mit ein paar Modifikationen übernommen werden. Da es keine `start` Assembler Funktion gibt, gibt es diese auch nicht in der Dispatcher Klasse. Und anstelle der `SwitchTo` Funktion hat der Dispatcher die `contextSwitch` Methode. Statt dass der Idle Thread vom Scheduler erstellt wird, wird diese am Anfang vom Boot an Stelle von der `main` gestartet. Der Idle Thread muss nicht initialisiert werden, da die Register direkt vor dem Start des Threads mit den richtigen Werten geladen werden können. Des Weiteren wird der Scheduler als Namespace umgesetzt, anstelle einer Klasse mit einem Singleton Objekts. Der Dispatcher wird ebenfalls als Namespace umgesetzt. Und der Inhalt der Funktionen wird an die restlichen Änderungen des Systems angepasst.

3.4.5 Eine multi-threaded Testanwendung

Um das Multithreading zu testen, sollen nun Testanwendungen geschrieben werden. Dazu sollte der HelloWorld Thread aus der Vorgabe genutzt werden. Dieser gibt, entsprechend der Aufgabenstellung, einen Text auf der Textausgabe aus und terminiert. Anschließend soll der Idel thread ausgeführt werden. Um den Test durchzuführen, sollen die beiden Threads in `main` mit dem Scheduler registriert werden. Dann soll der Scheduler mit `schedule` gestartet werden. [34]

Ein zweiter Test soll mehrere Threads parallel testen. Dazu sollen drei Zähler Threads von der Instanz `loopthread`, geschrieben werden. Ein Haupt-Thread soll die drei Zähler Threads erzeugen. Der Haupt-Thread soll eine gewisse Menge an `yield` durchführen, beispielsweise 1000, und dann mit Exit sich selbst terminieren. Bevor sich der Haupt-Thread beendet, soll er einen der Zähler Threads terminieren. [34]

Für das Testen der ARM-Umsetzung werden zwei Threads genutzt. Dabei handelt es sich um einen Haupt-Thread und einen Zähler Thread, welcher vom Haupt-Thread gestartet wird. Der Haupt-Thread selbst soll auch einen Zähler ausgeben, damit nachvollzogen werden kann, ob dieser läuft. Einmal soll der Zähler endlos laufen und einmal mit dem Beenden der `main` durch ein Return. Da ein Return-Wert für das Beenden der `main` eines Threads implementiert wurde, muss dies auch überprüft werden. Dazu wird der Debugger genutzt. Das erfolgreiche Beenden des Zähler Threads wird schrittweise überprüft. Damit der Aufruf einer Exit Funktion vermieden werden kann, wurde die Rücksprung-Adresse der `main` mit einer Exit Funktion geladen. Das heißt, wenn die `main` beendet wird, wird die Exit Funktion so aufgerufen, als ob diese von der `main` Funktion aufgerufen wurde. Da der Return Wert in `X0` abgelegt wird, ist der erste Parameter der Exit Funktion zu diesem Wert gesetzt. Exit löst dann einen SVC aus, der dann den Return Wert in dem Thread Objekt speichert und entfernt den Thread aus dem Scheduler. Danach wird in der Exit-Funktion mittels einer Endlosschleife auf den nächsten Threadwechsel gewartet. Für den Test muss damit das präemptive Multithreading aus dem nächsten Abschnitt implementiert sein. Der Haupt-Thread kann sich den Rückgabe-Wert mittels `join` aus der lib/thread.hpp holen. `join` nutzt SVC Aufrufe, um sich den Wert vom Betriebssystem zu holen und zu warten, falls der Thread noch läuft.

3.5 Präemptives Multithreading

In der Aufgabe Präemptives Multithreading soll das Betriebssystem durch die Aufgabe Präemptives Multitasking mit präemptives Context Switching erweitert und anschließend mit der Aufgabe Multithreading Anwendung getestet werden.

Die folgenden Lernziele sollen dabei laut Aufgabenstellungen erreicht werden.

- Tieferes Verständnis von präemptiven Multitasking
- CPU-Entzug mithilfe des PIT
- Synchronisierung des Schedulers und des Allokators gegenüber dem PIT-Interrupt [35]

3.5.1 Programmable Interval Timer (PIT)

Für präemptives Multithreading braucht es einen Timer. Dazu sollte bei x86 der Programmable Interval Timer (Pit) dienen. Der Timer soll das Umschalten von Threads erzwingen und eine Systemzeit als globale Variable inkrementieren. Der PIT soll auf ein 10 ms Intervall konfiguriert werden. Damit entsprechen 10 ms einem Tick. Zum Testen soll die Systemzeit alle 100 Ticks ausgegeben werden. [35]

Da ARM keinen PIT besitzt, wurde bei der Umsetzung der Generic Timer von ARM benutzt. Der Timer basiert auf dem System Counter. Dieser wird mit einer festen Frequenz zwischen 1 MHz und 50 MHz inkrementiert. In dieser Arbeit wird der Timer durch TVAL konfiguriert, siehe Kapitel Timer. Das TVAL Register wird also mit dem Intervall geladen und dann wird der Timer im Control Register aktiviert. Nach dem Auslösen eines Interrupts muss der TVAL Wert erneut geladen werden.

timer.hpp C++

```
namespace devices::Sytemtimer {
    namespace timer = registers::CNTV_CTL_EL0;
    namespace value = registers::CNTV_TVAL_EL0;
    namespace gic = peripherals::GIC;

    inline void init() {
        gic::configIrq(TIMER_IRQ, gic::trigger::edge);
        gic::setPriority(TIMER_IRQ, 0);
        gic::setTarget(TIMER_IRQ, gic::cpuTarget::interface0);
        gic::clearPending(TIMER_IRQ);
        gic::enableInt(TIMER_IRQ);
    }
    inline void rest() {
        timer::set(0);
        value::set(TIMESLOT_TIME);
        time++;
        timer::IMASK(false);
        timer::ENABLE(true);
    }
    inline void start() {
        init();
        rest();
    }
}
```

Außerdem muss der Timer interrupt bei GIC aktiviert werden und konfiguriert werden. Eine Systemzeit soll ebenfalls im Timer interrupt inkrementiert werden.

3.5.2 Threadumschaltung mithilfe des PIT

Nun soll der Timer genutzt werden, um die Threads präemptive umzuschalten. Dazu soll der Timer Interrupt bei jedem Tick die `preemptive` Methode vom Scheduler aufrufen. Damit ist eine Zeitscheibe ein Tick. Damit die Ready Queue nicht beschädigt wird, muss der Scheduler ebenfalls durch eine interrupt Sperre geschützt werden. Nach `thread_start` und `thread_switch` müssen Interrupts wieder eingeschaltet werden. Außerdem muss mit einer Instanzvariable sichergestellt werden, dass der Scheduler initialisiert wird. [35]

Bei der Umsetzung in ARM ruft der Timer Interrupt ebenfalls die präemptive Methode vom Scheduler auf. Eine Interrupt Sperre für den Scheduler wird nicht gebraucht, da Interrupts beim Eintreten in eine Exception bereits durch die Hardware deaktiviert werden und der Scheduler nur während einer Exception der Scheduler läuft. Außerdem muss der Scheduler nicht initialisiert werden, weil dieser keine Elemente zum Initialisieren hat.

3.5.3 Testanwendung mit Multithreading

Das präemptive Multithreading soll, laut Aufgabenstellung, getestet werden. Dazu soll ein Zähler Threads gestartet werden und ein Thread, der eine Melodie abspielt. Die Systemzeit soll außerdem im Interrupt ausgegeben werden. [35]

Da der PC Lautsprecher nicht umgesetzt wird, wird dieser Test stattdessen mit drei Zähler Threads durchgeführt.

3.6 Semaphore

In der Aufgabe Semaphore soll ermöglicht werden Threads untereinander zu synchronisieren. Dazu soll ein Semaphore für die Synchronisation implementiert werden.

Laut Aufgabenstellungen sollen damit die folgenden Lernziele erreicht werden.

- Verstehen wie Semaphoren implementiert werden und damit Thread-Synchronisierung funktioniert
- Erweitern des Schedulers sowie der Thread-Zustände, um Blockieren und Deblockieren zu realisieren [36]

3.6.1 Semaphore

Die Aufgabe gibt vor, dass ein Semaphore implementiert werden soll, damit die Threads sich gegenseitig synchronisieren können, ohne dass Interrupts deaktiviert werden müssen. Dazu sollte die Semaphore Klasse implementiert werden. Jedes Objekt soll eine Warteschlange besitzen, auf die die blockierten Threads zugreifen können, welche auf den `v` Aufruf warten. Die `p` und `v` Methoden müssen atomar sein. Dafür soll eine `Spinlock` Klasse sorgen. Diese ist bereits für den Kurs implementiert. Der Scheduler muss mit `block` und `deblock` erweitert werden. `block` soll auf den nächsten Thread schalten, ohne den aktuellen Threads in die ready Queue einzufügen. `deblock` soll den Thread wieder in die ready Queue einfügen. Die beiden Methoden sollen, laut Aufgabenstellung, mit einer interrupt Sperre versehen werden. [36]

Für die Umsetzung dieser Arbeit kann die Semaphore Klasse equivalent umgesetzt werden. Ebenfalls equivalent umgesetzt sind `block` und `deblock`. Dennoch werden ein paar Modifikationen an der Semaphore Klasse vorgenommen. Zum einen wird `p` zu `acquire` und `v` zu `release` unbenannt, damit die Klasse näher an der Benennung der Semaphore Klasse von Std ist und die Lesbarkeit erhöht wird. Außerdem wird anstelle eines Spinlock ein Mutex benutzt und es wird ein `lock_guard` implementiert. Der `lock_guard` ist ein Objekt, das den Mutex bei der Erstellung locked und beim Löschen des Objekts wieder freigibt. Der `lock_guard` wird zwecks Lesbarkeit am Anfang der Semaphore Methoden erstellt und muss nicht mehr am Ende der Methode released werden, da beim Ende der Methode der Destructor vom `lock_guard` aufgerufen wird.

Des Weiteren wird die Queue Klasse für die Warteschlange benutzt. Dies hat zur Folge, dass die Semaphor Klasse gegenüber den Interrupts nicht synchronisiert ist, da die Queue Klasse den Allocator benutzt. Die Interrupts haben ihren eigenen Allocator benötigt und nur Threads hätte es erlaubt sein dürfen, den Allocator von `new` und `delete` zu nutzen. Der Allocator wurde jedoch in Speicherverwaltung, als Namespace und die Warteschlange ohne ein Allocator Template umgesetzt. Das heißt, das Synchronisieren der Semaphor Klasse gegenüber Interrupts hätte einige Änderungen an dem Allocator als auch an der Warteschlange zur Folge gehabt, welche aus Zeitgründen nicht umgesetzt werden.

Statt eines Spinlock wird ein Mutex umgesetzt. Jedoch wird die Warte Instruktion auskommentiert, da die Umsetzung über den Umfang der Arbeit hinausginge.

Für die Umsetzung des Mutex werden spezielle Instruktionen benutzt, um die Race Condition mit anderen Kernen zu vermeiden. Dies erfolgt anstelle der `lock` Instructions in x86. Der folgende Code ist eine Abwandlung einer ARM Mutex Implementation für ARMv6 M-Serie. [37]

mutex.hpp C++

```
namespace lib {
    class mutex {
        // ...
        inline void lock() {
            __asm __volatile__(
                R"(     mov     x1, #0x01       // write constant 1 to x1 as lock value
                  1:
                      ldxr    x2, [  %[value] ]
                      cmp     x2, x1          // Test if mutex is locked or unlocked
                        beq     2f            // If locked - wait for it to be released, from 2
                      stxr    w2, x1, [ %[value] ]  // Not locked, attempt to lock it
                      cmn     w2, #0x01       // Check if Store-Exclusive failed
                        beq     1b            // Failed - retry from 1
                      dmb ish                 // Required before accessing protected resource
                      b       3f
                  2:  // Take appropriate action while waiting for mutex to become unlocked
                      //svc     0x10            // signal the kernel that this threads is waiting on lock
                      b       1b              // Retry from 1
                  3:  // end
                      CLREX
                )"
                : /*no out*/
                : [value] "r" (&value)
                : "x1", "x2");
        }
        // ...
    }
}
```

Der Befehl `ldxr` markiert die Speicheradresse exklusiv für den Kern, auf welche zugegriffen wird. Auf diese Weise gibt der Befehl `strxr` einen Fehlschlag zurück, wenn ein anderer Kern auf diese Adresse zugreift.

Kapitel 4

Verification

Die Funktionalität des Codes wurde durch das Schreiben kurzer Tests und manuelles Überprüfen kontrolliert. Dementsprechend wurde der Code an kritischen Stellen mit dem Debugger schrittweise durchgegangen und überprüft, ob das Verhalten und die Werte in den Registern dem gewünschten Verhalten entsprechen.

Um das System im Ganzen zu testen, wurden zum Abschluss zwei Test Threads und ein Konsolen Thread programmiert. Der Konsolen Thread wartet auf eine Eingabe durch den Nutzer. Der Nutzer kann durch einen Befehl entweder einen Zähler Thread oder einen Semaphore Thread starten. Der Konsolen Thread testet das Starten und Beenden eines Threads, sowie die Rückgabe von Rückgabewerte. Der Zähler Thread zählt bis 5 hoch und gibt jede Zahl durch die Textausgabe aus. Danach beendet der Thread sich selbst und gibt eine festen Wert zurück, dieser wird von dem Konsolen Thread ausgegeben. Der Semaphore Thread ist ein Thread, welcher eine Lösung des Erzeuger-Verbraucher-Problems darstellt. Dadurch soll das Ablaufen mehrerer Threads und die Synchronisation dieser, durch eine Semaphore, getestet werden.

Kapitel 5

Fazit

Ziel der Arbeit war es, ein Betriebssystem für die ARM-Architektur auf Basis des Masterkurses Betriebssystem-Entwicklung zu entwickeln und damit die Verwirklichung einer ARM-basierten Lösung für die einzelnen Aufgaben des Kurses.

Ein Teil der Umsetzung des Kurses konnte im Zuge dieser Arbeit in ARM äquivalent realisiert werden. Ein anderer Teil musste aufgrund der Besonderheiten der ARM-Architektur anders realisiert werden. Zudem wurden einige optionale Änderungen vorgenommen. Diese sind unabhängig von der Architektur und führten zu Vorteilen in der Umsetzung der jeweiligen Aufgabe.
Für die Ausgabe von Bildschirminhalten wurde, statt des CGAs von x86, ein UART umgesetzt. Für die Eingabe wurde anstelle der PS/2 Tastatur ebenfalls ein UART genutzt. Die Speicherverwaltung ist unabhängig von der Architektur und konnte äquivalent umgesetzt werden. Statt Klassen wurden Namespaces verwendet. Da ARM keinen PIC hat, wurde für die Interrupt Verarbeitung der GIC verwendet. Die Interrupt Weiterleitung konnte äquivalent umgesetzt werden. Jedoch wurden optionale Änderungen durchgeführt. Koroutinen hätten realisiert werden können, wurden jedoch ausgelassen. Stattdessen wurden direkt Threads umgesetzt. Die Thread Umschaltung musste an die Register von ARM angepasst werden. Welche Architektur benutzt wurde, hatte auf dem Scheduler keine Auswirkung. Dementsprechend wurde der Scheduler äquivalent umgesetzt. Da ein PIT in ARM nicht existiert, wurde der Generic Timer umgesetzt, welcher die gleichen Aufgaben erfüllt. Der Semaphore wurde äquivalent realisiert, mit der Ausnahme der Race Condition in der Spinlock Klasse. Diese musste an die Architektur angepasst werden. Im Hinblick auf die Lösung des Kurses, hat die Umsetzung dieser Arbeit geringe Einschränkungen. So wurde keine Tonausgabe verwendet. Es fehlt ein Tastatur Interrupt und die Interrupt Sperre.

Das Ergebnis der Arbeit ist ein lauffähiges Betriebssystem mit Lösungen für die einzelnen Aufgaben des Masterkurses Betriebssystem-Entwicklung. Damit wurde das Ziel der Arbeit erreicht. Die fehlenden Bestandteile wie Ton und Tastatur Interrupt stellen keine wesentliche Einschränkung des Betriebssystems dar. Eine kleine Einschränkung ist das Fehlen der Interrupt Sperre. Dies hat zur Folge, dass die Semaphore Klasse nicht sicher gegenüber Interrupts ist. Die Semaphore Klasse benötigt, wie auch der Kernel, einen Allocator. Jedoch ist es nicht möglich, beide mit einer Interrupt Sperre zu synchronisieren. Um dies zu lösen, müssten zwei separate Allocator erstellt werden, eine

für den Kernel und einen für den User Bereich. Für die Umsetzung des Betriebssystems stellt dies lediglich eine kleine Einschränkung dar, da die Nutzung einer Interrupt Sperre generell vermieden werden sollte, weil diese zu einem Deadlock führen kann.

Um die wachsenden Bedeutung der ARM-Architektur im Masterkurs Betriebssystem-Entwicklung zu berücksichtigen, sollte der Kurs um die ARM-Architektur erweitert werden. Dazu sollten die architektur-bedingten Unterschiede in den Aufgaben des Kurses berücksichtigt werden. Hierfür sind Anpassungen am Kurs und den Lösungen der Arbeit notwendig. In dieser Arbeit wurde die Ein- und Ausgabe mittels UART umgesetzt. Der in der Arbeit verwendet UART PL011 ist für die ARM-Architektur spezifisch. Des Weiteren fehlt durch die Nutzung eines UART die Möglichkeit, Text an festen Positionen auf dem Bildschirm ausgeben zu können. Um die Aufgabe anzugleichen und Text an festen Positionen ausgeben zu können, könnte die Umsetzung der Ein- und Ausgabe mittels des Framebuffer "ramfb" und der "virto-keyboard-pic" Tastatur erfolgen. Qemu ist in der Lage, beide Geräte für beide Architekturen zu simulieren. Da die Interrupt Verarbeitung, der Timer und der Dispatcher Architektur spezifisch sind, könnte der Kurs mit zusätzlichen Unteraufgaben, welche die ARM spezifischen GIC, Generic Timer und Dispatcher behandeln, angepasst werden. Interrupt Sperren, die innerhalb eines Interrupts ausgelöst werden, müssen nicht angepasst werden. Interrupt Sperren, die innerhalb des Thread Codes ausgelöst werden sollen, sollten vermieden werden. Dies kann durch das Verzichten von kritischen Abschnitten, welche eine Synchronisierung gegenüber den Interrupts benötigen, erreicht werden. In der Aufgabe Semaphore sollte, analog zu den x86 Aufgabenstellungen, eine Spinlock-Klasse für ARM bereitgestellt werden. Statt direkt Threads umzusetzen, sollte die Lösung dieser Arbeit um Koroutinen erweitert werden, sodass der Unterschied zwischen Koroutinen und Threads auch in der Lösung für die ARM-Architektur gezeigt werden kann. Die offiziellen Dokumentationen von Qemu und ARM waren, im Bezug auf den Inhalt dieser Arbeit, unzureichend. Fehlende Informationen könnten im Kurs zur Verfügung gestellt werden. Sollte eine Einbindung der ARM Lösungen den Kurs zu umfangreich werden lassen, so könnten Unteraufgaben wie die Tonausgabe entfallen oder es könnte eine gesteigerte Anzahl an Hilfsmitteln, wie zum Beispiel die vorgegebene Spinlock-Klasse, zur Verfügung gestellt werden.

Auffällig war das Fehlen von Paging und Prozessen im Kurs. Eine Erweiterung um diese Elemente würde den Kurs vervollständigen. Durch die Nutzung des Generic Loader von Qemu wäre es möglich Programme in den RAM zu laden und diese anschließend als Prozesse auszuführen. Für die Umsetzung von Prozessen wird zusätzlich Paging benötigt, da die Speicheradressen der Programme alle in Relation zu ihrer Anfangsadressen im Speicher abgelegt werden. Der Generic Loader ist sowohl für die x86-Architektur als auch für die ARM-Architektur vorhanden. Eine Umsetzung in x86 und ARM sowie deren Integration in den Masterkurs könnte Thema einer weiteren Arbeit werden.

Literatur

[1] "The Future is Built on Arm". (), Adresse: https://www.arm.com/company (besucht am 19. 09. 2023).

[2] "Supermicro, der weltweite drittgrößte Anbieter von Servern, erweitert seine Unternehmenszentrale im Silicon Valley". (), Adresse: https://www.supermicro.com/de/pressreleases/ranked-3rd-largest-supplier-servers-world-supermicro-expands-its-silicon-valley (besucht am 19. 09. 2023).

[3] "Supermicro Adds ARM-based Servers using Ampere® Altra® and Ampere Altra® Max Processors targeting Cloud-Native Applications". (), Adresse: https://www.supermicro.com/de/pressreleases/supermicro-adds-arm-based-servers-using-amperer-altrar-and-ampere-altrar-max (besucht am 19. 09. 2023).

[4] "Die nächste Generation des Mac". (), Adresse: https://www.apple.com/de/newsroom/2020/11/introducing-the-next-generation-of-mac/ (besucht am 19. 09. 2023).

[5] "Apple stellt M2 Ultra vor". (), Adresse: https://www.apple.com/de/newsroom/2023/06/apple-introduces-m2-ultra/ (besucht am 19. 09. 2023).

[6] "Global market share held by operating systems for desktop PCs, from January 2013 to July 2023". (), Adresse: https://www.statista.com/statistics/218089/global-market-share-of-windows-7/ (besucht am 19. 09. 2023).

[7] "Distribution of Intel and AMD x86 computer central processing units (CPUs) worldwide from 2012 to 2023, by quarter". (), Adresse: https://www.statista.com/statistics/735904/worldwide-x86-intel-amd-market-share/ (besucht am 19. 09. 2023).

[8] "Arm-Architektur". (2023), Adresse: https://de.wikipedia.org/wiki/Arm-Architektur (besucht am 19. 09. 2023).

[9] "ARM Overview". (2023), Adresse: https://wiki.osdev.org/ARM_Overview (besucht am 19. 09. 2023).

[10] "Privilege and Exception levels". (), Adresse: https://developer.arm.com/documentation/102412/0103/Privilege-and-Exception-levels (besucht am 19. 09. 2023).

[11] "The TrustZone hardware architecture". (), Adresse: https://developer.arm.com/documentation/100935/0100/The-TrustZone-hardware-architecture- (besucht am 19. 09. 2023).

[12] "Exception types". (), Adresse: https://developer.arm.com/documentation/102412/0103/Exception-types?lang=en (besucht am 19. 09. 2023).

[13] "AArch64 exception vector table". (), Adresse: https://developer.arm.com/documentation/100933/0100/AArch64-exception-vector-table (besucht am 19. 09. 2023).

[14] "ARMv8 Registers". (), Adresse: https://developer.arm.com/documentation/den0024/a/ARMv8-Registers (besucht am 19. 09. 2023).

[15] "Parameters in general-purpose registers". (), Adresse: https://developer.arm.com/documentation/den0024/a/The-ABI-for-ARM-64-bit-Architecture/Register-use-in-the-AArch64-Procedure-Call-Standard/Parameters-in-general-purpose-registers (besucht am 19. 09. 2023).

[16] "NEON and floating-point registers". (), Adresse: https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/NEON-and-floating-point-registers (besucht am 19. 09. 2023).

[17] "AArch64 special registers". (), Adresse: https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/AArch64-special-registers (besucht am 19. 09. 2023).

[18] "Saved Process Status Register". (), Adresse: https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/AArch64-special-registers/Saved-Process-Status-Register (besucht am 19. 09. 2023).

[19] "Exception Link Register (ELR)". (), Adresse: https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/AArch64-special-registers/Exception-Link-Register--ELR- (besucht am 19. 09. 2023).

[20] "SVC". (), Adresse: https://developer.arm.com/documentation/dui0489/i/arm-and-thumb-instructions/svc (besucht am 19. 09. 2023).

[21] "The Generic Interrupt Controller". (), Adresse: https://developer.arm.com/documentation/den0024/a/AArch64-Exception-Handling/The-Generic-Interrupt-Controller (besucht am 19. 09. 2023).

[22] "Special interrupt numbers". (), Adresse: https://developer.arm.com/documentation/ihi0048/b/Interrupt-Handling-and-Prioritization/General-handling-of-interrupts/Special-interrupt-numbers (besucht am 19. 09. 2023).

[23] "AArch64 Programmer's Guides Generic Timer". (2019), Adresse: https://tc.gts3.org/cs3210/2020/spring/r/aarch64-generic-timer.pdf (besucht am 19. 09. 2023).

[24] "Functional description". (), Adresse: https://developer.arm.com/documentation/ddi0183/g/functional-overview/functional-description (besucht am 19. 09. 2023).

[25] "Control Register, UARTCR". (), Adresse: https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/line-control-register--uartlcr-h?lang=en (besucht am 19. 09. 2023).

[26] "Line Control Register, UARTLCR_H". (), Adresse: https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/line-control-register--uartlcr-h?lang=en (besucht am 19. 09. 2023).

[27] "Flag Register, UARTFR". (), Adresse: https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/flag-register--uartfr?lang=en (besucht am 19. 09. 2023).

[28] "Data Register, UARTDR". (), Adresse: https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/data-register--uartdr?lang=en (besucht am 19. 09. 2023).

[29] "Modulhandbuch für den MasterStudiengang Informatik". (2023), Adresse: https://www.cs.hhu.de/fileadmin/redaktion/Fakultaeten/Mathematisch-Naturwissenschaftliche_Fakultaet/Informatik/Studium/Allgemeines/modulhandbuch_master_de.pdf (besucht am 19. 09. 2023).

[30] "Betriebssystementwicklung (Master)". (2023), Adresse: https://coconucos.cs.uni-duesseldorf.de/lehre/home/lectures/bse/overview (besucht am 19. 09. 2023).

[31] "Aufgabe 1: Ein-/Ausgabe". (2023), Adresse: https://github.com/hhu-bsinfo/hhuTOSc/tree/aufgabe-1 (besucht am 19. 09. 2023).

[32] "Aufgabe 2: Speicherverwaltung und PC-Speaker". (2023), Adresse: https://github.com/hhu-bsinfo/hhuTOSc/tree/aufgabe-2 (besucht am 19. 09. 2023).

[33] "Aufgabe 3: Interrupts". (2023), Adresse: https://github.com/hhu-bsinfo/hhuTOSc/tree/aufgabe-3 (besucht am 19. 09. 2023).

[34] "Aufgabe 4: Koroutinen und Threads". (2023), Adresse: https://github.com/hhu-bsinfo/hhuTOSc/tree/aufgabe-4 (besucht am 19. 09. 2023).

[35] "Aufgabe 5: Präemptives Multithreading". (2023), Adresse: https://github.com/hhu-bsinfo/hhuTOSc/tree/aufgabe-5 (besucht am 19. 09. 2023).

[36] "Aufgabe 6: Semaphore". (2023), Adresse: https://github.com/hhu-bsinfo/hhuTOSc/tree/aufgabe-6 (besucht am 19. 09. 2023).

[37] "Implementing a mutex". (), Adresse: https://developer.arm.com/documentation/dht0008/a/arm-synchronization-primitives/practical-uses/implementing-a-mutex (besucht am 19. 09. 2023).

Abbildungsverzeichnis

Tabellenverzeichnis

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 11. Oktober 2023

Markus Schäfer