Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

Praktikum Systemprogrammierung

Versuch 2

Scheduler / Stack

Lehrstuhl Informatik 11 - RWTH Aachen

6. April 2021

Commit: dd0ebb7b

Inhaltsverzeichnis

Dieses Dokument ist Teil der begleitenden Unterlagen zum *Praktikum Systemprogrammierung*. Alle zu diesem Praktikum benötigten Unterlagen stehen im Moodle-Lernraum unter https://moodle.rwth-aachen.de zum Download bereit. Folgende E-Mail-Adresse ist für Kritik, Anregungen oder Verbesserungsvorschläge verfügbar:

support.psp@embedded.rwth-aachen.de

2 Scheduler / Stack

Zu den Kernaufgaben eines Multitasking-Betriebssystems gehören die Verwaltung von Programmen und Prozessen, das Prozess-Scheduling, sowie die Verwaltung des verfügbaren Speichers und weiterer Betriebsmittel. Der Scheduler ist die Komponente des Betriebssystems, die in regelmäßigen Zeitabständen den derzeit ausgeführten Prozess unterbricht, um die Prozessorzeit neu zuzuteilen.

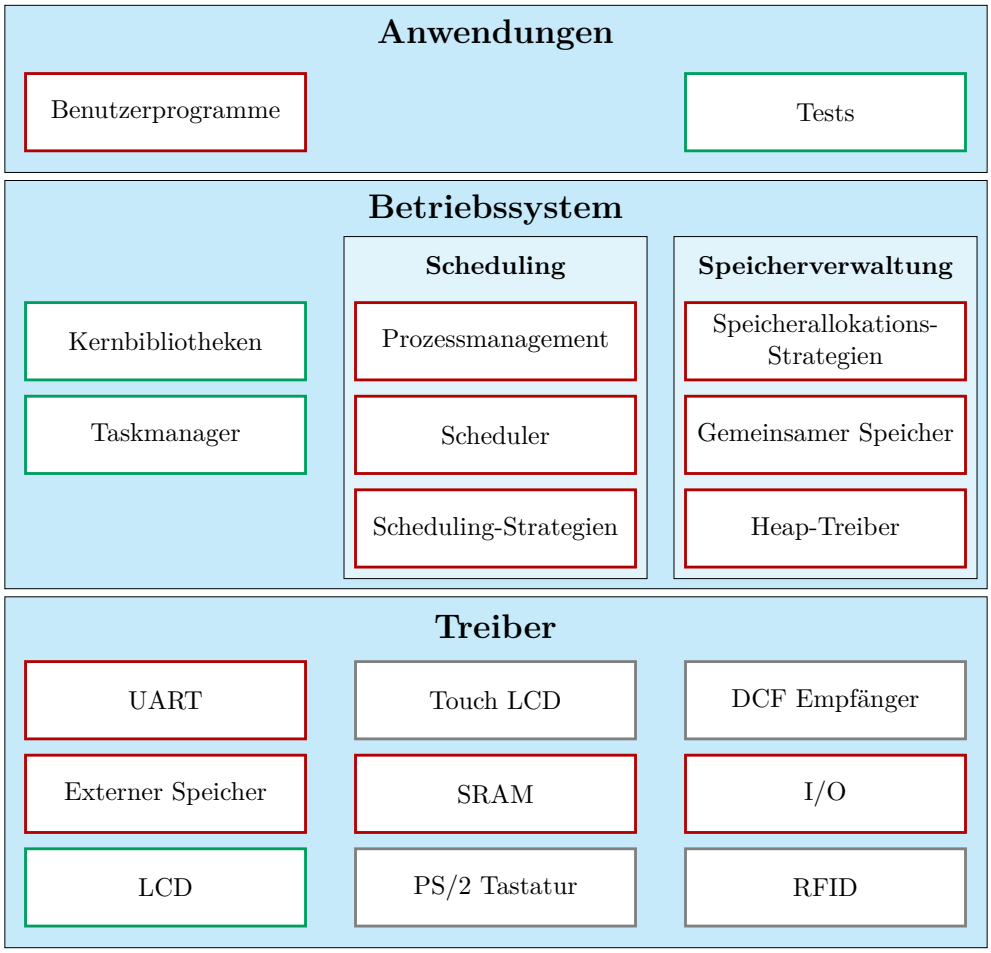

Abbildung 2.1: Schichtenmodell des SPOS-Betriebssystems

Dieser und folgende Versuche behandeln die Implementierung des Betriebssystems SPOS für das aus Versuch 1 bekannte Evaluationsboard. In Abbildung 2.1 ist die Architektur

des Betriebssystems visualisiert. Der Kern von SPOS umfasst Scheduling und Speicherverwaltung, die auf den Kernbibliotheken aufbauen und durch den Taskmanager überwacht werden können. Ferner wird SPOS durch Treiber erweitert, die zum Beispiel die Ansteuerung eines LCDs ermöglichen. Anwendungen, die innerhalb von SPOS ausgeführt werden, können vom System bereitgestellte Funktionen nutzen.
Im Diagramm grün markierte Komponenten sind vorgegeben. Komponenten mit einem roten Rahmen werden im Laufe des Praktikums implementiert. Einer der grau markierten Treiber wird Bestandteil des letzten Versuchs sein.

2.1 Versuchsinhalte

In diesem Versuch wird der Scheduler implementiert, der das quasi-gleichzeitige Ausführen mehrerer Prozesse ermöglicht. In Ergänzung dazu werden grundlegende Funktionalitäten und Datenstrukturen eines Betriebssystems implementiert, um ein erstes, rudimentäres Prozess-Scheduling zu ermöglichen.

Achtung

Ihre Programmcodes dieses und aller folgenden Versuche bauen aufeinander auf. Daher ist es wichtig, sorgfältig zu arbeiten und sich exakt an die Vorgaben aus dieser Versuchsbeschreibung zu halten.

2.2 Lernziel

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Funktionsweise eines Schedulers
- Verwaltung von kritischen Bereichen
- Verwaltung von Betriebssystem- und Prozessstacks
- Verwaltung von Programmen und Prozessen
- Fehlererkennung

2.3 Grundlagen

Es ist notwendig, vor Bearbeitung der Hausaufgaben das gesamte Grundlagenkapitel zu lesen, da die zu implementierenden Komponenten sehr eng miteinander verflochten sind. Im Folgenden werden zunächst die Konzepte „Programme und Prozesse“, „Scheduler“, „kritische Bereiche“ und „Stack“ vorgestellt. Nach der allgemeinen Darstellung dieser Konzepte wird auf die spezifische Umsetzung im Rahmen von SPOS eingegangen.

2.3.1 Programme und Prozesse

Ein Betriebssystem unterscheidet zwischen Programmen und Prozessen. Ein Programm ist eine endliche Folge von Operationen. Ein Prozess ist eine Instanz eines Programms, welche durch das Betriebssystem mit dessen sogenanntem *Laufzeitkontext* verwaltet wird. Der Laufzeitkontext ist die Summe der Prozessorregister, die zu einem Zeitpunkt den Ausführungszustand eines Prozesses abbilden. Um in einem Betriebssystem ein Programm mehrfach starten zu können, wird für jede Programminstanz ein Prozess mit eigenem Laufzeitkontext angelegt. Somit wird jede Programminstanz separat voneinander ausgeführt. Ein solches Betriebssystem nennt man auch Multitasking-Betriebssystem.
In der Regel wird zwischen zwei Arten von Programmen unterschieden: Terminierende und nicht-terminierende Programme. Der Einfachheit halber werden in diesem Versuch nur nicht-terminierende Programme, und somit auch Prozesse, behandelt. Die Behandlung von terminierenden Programmen/Prozessen wird in Versuch 3 (*Heap / Schedulingstrategien*) erläutert.

2.3.2 Scheduler

Ein Scheduler wird in Multitasking-Betriebssystemen eingesetzt, um das Betriebsmittel Prozessorzeit zu verwalten und den Wechsel zwischen Prozessen transparent durchzuführen. Transparenz bedeutet in diesem Fall, dass jedem Prozess Prozessorzeit und Stackspeicher idealerweise so zur Verfügung stehen, dass der Prozess nicht unterscheiden kann, ob er exklusiven Zugriff auf das System hat oder nicht.
Ein Scheduler kann beispielsweise das aktuell ausgeführte Programm in periodischen Zeitabständen unterbrechen, um einen anderen Prozess auszuführen. Wenn ein Prozess vom Scheduler unterbrochen wird, muss der Laufzeitkontext dieses Prozesses gespeichert werden. Sobald der Prozess zu einem späteren Zeitpunkt wieder aktiviert und Rechenzeit zugewiesen wird, muss der entsprechende Laufzeitkontext wiederhergestellt werden, um sicherzustellen, dass der Prozess in dem Zustand fortgesetzt wird, in dem er unterbrochen wurde. Da der Zustand exakt wieder hergestellt wird, kann der Prozess also nicht erkennen, dass er unterbrochen wurde.
Falls kein Prozess zur Ausführung zur Verfügung steht wird ein sogenannter *Leerlaufprozess* ausgewählt. Dieser Systemprozess verbraucht innerhalb des Betriebssystems ungenutzte Prozessorzeit.

Schedulingstrategien

Um festzustellen, welcher Prozess bei einem Prozesswechsel Rechenzeit erhalten soll, wird vom Scheduler eine sogenannte *Schedulingstrategie* verwendet. Eine Schedulingstrategie sucht aus der Menge der im Betriebssystem laufenden Prozesse nach bestimmten Kriterien den Prozess aus, der als nächstes ausgeführt werden soll. Einem Scheduler stehen in der Regel mehrere verschiedene Schedulingstrategien zur Verfügung, zwischen denen beliebig gewählt werden kann. Als mögliche Strategien werden in diesem Versuch die Strategien *Random*, *Even*, *RunToCompletion*, *RoundRobin* und *InactiveAging* betrachtet.
Für jede Strategie gilt, dass der Leerlaufprozess nur dann ausgewählt werden soll, falls kein anderer Prozess auf Ausführung wartet. Dies ist insbesondere dann relevant, falls ein Prozess gestartet wird, nachdem kein Prozess aktiv war.
Im Folgenden werden die Strategien vorgestellt, die in SPOS benutzt werden sollen:

Even Die *Even*-Strategie durchläuft die auswählbaren Prozesse in aufsteigender Reihenfolge, sodass jeder Prozess gleichmäßig oft ausgewählt wird und auf den letzten auswählbaren Prozess wieder der erste folgt.

Random Die *Random*-Strategie wählt zufällig einen der auswählbaren Prozesse gemäß der diskreten Gleichverteilung in konstanter Laufzeit aus.

RunToCompletion Die *RunToCompletion*-Strategie wählt den aktuell laufenden Prozess so lange aus, bis dieser terminiert. Anschließend wird der nächste auswählbare Prozess analog zur *Even*-Strategie ausgewählt. In diesem Versuch terminieren Programme noch nicht, jedoch wird dies in späteren Versuchen möglich sein.

RoundRobin Bei der *RoundRobin*-Strategie verwaltet der Scheduler eine Zeitscheibe für jeden aktuell laufenden Prozess. Diese Zeitscheibe wird mit der Priorität `priority` des Prozesses initialisiert und bei jedem weiteren Scheduleraufruf dekrementiert, wodurch der Prozess `priority`-mal ausgewählt wird; also bis die Zeitscheibe den Wert `0` erreicht. In diesem Fall wählt der Scheduler den nächsten auswählbaren Prozess analog zur *Even*-Strategie aus.

InactiveAging Zusätzlich zur Priorität erhält jeder Prozess bei dieser Strategie ein Alter, das mit dessen Priorität initialisiert wird. Es wird immer der älteste Prozess als aktiver Prozess ausgewählt. Mit jedem Aufruf des Schedulers wächst das Alter jedes wartenden Prozesses um dessen Priorität (das Alter des aktiven Prozesses bleibt unverändert). Diese Strategie kann schrittweise beschrieben werden:

1. Das Alter aller inaktiven Prozesse wird um deren Priorität erhöht
2. Der älteste Prozess wird ausgewählt und dessen Alter auf dessen Priorität zurückgesetzt. Existieren mehrere Prozesse mit dem gleichen Alter, wird der höchstprio-

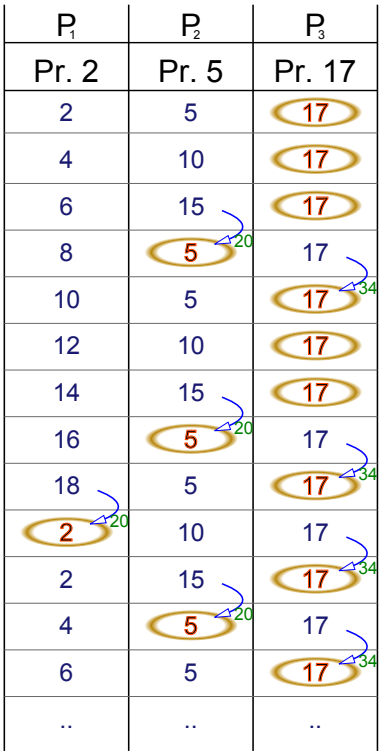

Abbildung 2.2: Veranschaulichung der Schedulingstrategie InactiveAging

risierte ausgewählt. Existieren mehrere Prozesse mit gleichem Alter und Priorität, wird der Prozess mit der kleinsten Prozess-ID ausgewählt.

Die Strategie *InactiveAging* wird in Abbildung 2.2 am Beispiel dreier Prozesse mit den Prioritäten 2, 5 und 17 verdeutlicht. Die zum jeweiligen Zeitpunkt aktiven Prozesse sind mit einem Kreis markiert. Die Zahlen in den Zellen geben das Alter des jeweiligen Prozesses an, nachdem der Scheduler ausgeführt wurde. Durch Pfeile wird angedeutet, dass ein Prozess das Alter des aktiven Prozesses überschritten hat und damit als nächster aktiver Prozess ausgewählt wird. Die Werte an den Pfeilen geben das neue Alter des jeweiligen Prozesses vor dem Zurücksetzen auf die entsprechende Priorität an.

Lernerfolgsfragen

- Angenommen in Abbildung 2.2 gäbe es einen vierten Prozess mit Priorität 85. Wie oft würde dieser maximal hintereinander ausgeführt?
- Wofür wird der Leerlaufprozess benötigt?

2.3.3 Kritische Bereiche

Betriebssysteme besitzen viele unterschiedlicher Betriebsmittel. Diese können zum Beispiel Rechenzeit, Datenstrukturen, Schnittstellen oder Geräte wie Drucker oder ein Bildschirm sein. Prozesse teilen sich den Zugriff auf die Betriebsmittel. Es gibt Operationen, in denen ein Prozess ein Betriebsmittel für eine Zeitspanne exklusiv nutzen muss, was bedeutet, dass kein anderer Prozess in dieser Zeit auf dieses zugreifen darf. Eine derartige Nutzung, bzw. die Dauer dieser Nutzung, wird als kritischer Bereich bezeichnet. Typischerweise werden kritische Bereiche verwendet, falls aus der gleichzeitigen Nutzung eines Betriebsmittels durch mehrere Prozesse ein inkonsistenter Zustand des Systems resultieren würde.

Prozess 1	Prozess 2
`lcd_writeChar('H')`	
`lcd_writeChar('a')`	
	`lcd_writeChar('W')`
	`lcd_writeChar('e')`
`lcd_writeChar('l')`	
`lcd_writeChar('l')`	
	`lcd_writeChar('l')`
	`lcd_writeChar('t')`
`lcd_writeChar('o')`	

(a) Prozessablauf ohne kritische Bereiche.
Ausgabe: `HaWelllto`

Prozess 1	Prozess 2
`os_enterCriticalSection()`	
`lcd_writeChar('H')`	
`lcd_writeChar('a')`	
`lcd_writeChar('l')`	
`lcd_writeChar('l')`	
`lcd_writeChar('o')`	
`os_leaveCriticalSection()`	
	`os_enterCriticalSection()`
	`lcd_writeChar('W')`
	`lcd_writeChar('e')`
	`lcd_writeChar('l')`
	`lcd_writeChar('t')`
	`os_leaveCriticalSection()`

(b) Prozessablauf mit kritischen Bereichen.
Ausgabe: `HalloWelt`

Abbildung 2.3: Kritische Bereiche zur Vermeidung einer Prozessverzahnung.

Abbildung 2.3 zeigt ein Beispiel für einen inkonsistenten Zustand, der durch zeitgleichen Zugriff zweier Prozesse auf das Betriebsmittel LCD entstanden ist. Im Prozessablauf in Abbildung 2.3(a) wollen beide Prozesse das LCD nutzen, um Text auszugeben. Der Einfachheit halber wird angenommen, dass der Scheduler den aktiven Prozess immer nach der Ausgabe von zwei Buchstaben wechselt. Die resultierende Ausgabe ist in dem Sinne inkonsistent, als dass eine unerwartete und ungewollte *Verzahnung* beider Zeichenketten erfolgt. In Abbildung 2.3(b) wird vor der Ausgabe des Textes von beiden Prozessen jeweils ein kritischer Bereich betreten, der das Betriebsmittel LCD für alle anderen Prozesse sperrt und erst wieder freigibt, wenn der jeweilige kritische Bereich verlassen wurde.

2.3.4 Stack

Ein Stack (Keller- oder Stapelspeicher) speichert Daten nach dem LIFO-Prinzip (Last-In-First-Out). Das bedeutet, dass das Byte, welches zuletzt abgespeichert wurde, als

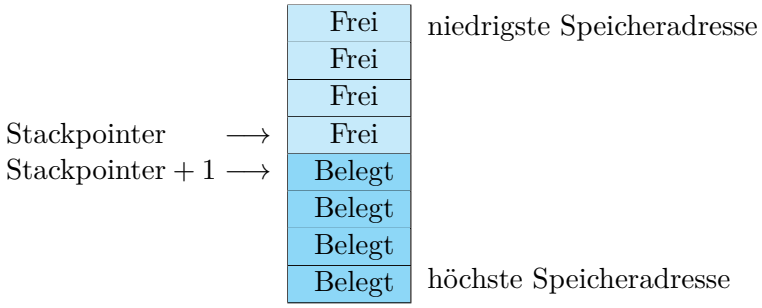

Abbildung 2.4: Beispiel für einen Stack.

erstes wieder gelesen wird. Auf dem Stack wird für gewöhnlich mit den Operationen „push“ (Ablegen eines neuen Byte oben auf dem Stack) und „pop“ (Auslesen und Herunternehmen des obersten Byte vom Stack) gearbeitet.
Der Zugriff auf einen Stack erfolgt nicht explizit über *push* und *pop*, sondern über einen sogenannten *Stackpointer*, der auf den ersten freien Platz des Speichers zeigt und entsprechend in- bzw. dekrementiert wird. Wird ein neues Byte auf den Stack gelegt, so wird es in der Speicherzelle abgelegt, auf die der Stackpointer zeigt. Danach wird der Stackpointer auf die nächstkleinere (freie) Speicherzelle verschoben. Abbildung 2.4 zeigt exemplarisch einen Stack mit einer maximalen Kapazität von acht Bytes und einem aktuellen Füllstand von vier Bytes.

Prozessstacks

In einem Betriebssystem werden Stacks in Form von *Prozessstacks* zur Speicherung lokaler Variablen und Rücksprungadressen verwendet. Das Betriebssystem verwaltet für jeden Prozess einen dieser Stacks und ermöglicht ihm so einen unabhängigen Programmfluss.
Das Prozessorregister `PC` (*Program Counter* - dt. Programmzähler) ist ein Pointer auf die nächste Instruktion, die vom Prozessor ausgeführt werden soll. Beim Sprung in eine Unterfunktion wird der aktuelle Wert des `PC`-Registers auf den Stack des laufenden Prozesses gelegt und somit zwischengespeichert, da nun die Adresse der ersten Instruktion der aufgerufenen Methode in das `PC`-Register geschrieben wird (man will diese schließlich nun ausführen). Während der Abarbeitung dieser Unterfunktion werden etwaige lokale Variablen auf dem Stack gespeichert, und gegen Ende der Funktion wieder heruntergenommen. Der Compiler erzeugt diese Instruktionen durch Analyse, wann lokale Variablen zuletzt verwendet werden, normalerweise selbstständig direkt in Maschinencode. Nach der letzten Instruktion der Unterfunktion wurden alle lokalen Variablen wieder abgebaut, und das oberste, was nun auf dem Prozessstack liegt, ist wieder die beim Hinsprung auf ihm abgelegte Rücksprungadresse. Der Compiler legt nach jedem `return` oder Ende einer `void`-Funktion Instruktionen ab, die die Rücksprungadresse vom Stack nehmen und wieder ins `PC`-Register laden.
So werden verschachtelte Funktionsaufrufe ermöglicht: Der Stack *wächst* beim Hinsprung

in Funktionen, und *baut* sich bei der Abwicklung der Verschachtelung, den Rücksprüngen, wieder *ab*.

Stack Overflows

Es ist möglich, dass der Stack eines Prozesses durch sehr viele ineinander verschachtelter Unterfunktionen oder die Definition vieler lokaler Variablen über die vorgesehene Größe hinauswächst. Die Werte, die dabei auf den Prozessstack gelegt werden, überschreiben möglicherweise einen anderen Prozessstack oder Datenbereich. Dies wird als *Stack Overflow* bezeichnet und gehört zur Gruppe der *Stackinkonsistenzen*.

Lernerfolgsfragen

- Wie werden Daten auf einem Stack gespeichert?
- Was bedeutet LIFO?
- Warum ist es nötig, jedem Prozess seinen eigenen Prozessstack zur Verfügung zu stellen, anstatt einen Stack für alle Prozesse zu verwenden?
- Welches (Fehl-)Verhalten des Programms ist möglich, wenn ein Stack Overflow aufgetreten ist?

Heuristik zur Erkennung von Stackinkonsistenzen

Die Konsistenz eines Prozessstacks kann mithilfe einer *Prüfsumme* bestimmt werden: Wenn ein Prozess einen Stack Overflow verursacht und Daten auf dem Prozessstack eines anderen Prozesses überschreibt, ändert sich mit hoher Wahrscheinlichkeit die Prüfsumme der im Speicherbereich des überschriebenen Prozessstacks befindlichen Daten.
Um die Konsistenz eines Prozessstacks zu gewährleisten, kann dessen Prüfsumme bei der Unterbrechung des zugehörigen Prozesses durch den Scheduler gebildet werden. Bevor der Scheduler diesem Prozess erneut Rechenzeit zuweist, wird für seinen Prozessstack erneut die Prüfsumme bestimmt. Ist die Prüfsumme mit dem Wert identisch, der bei der letzten Unterbrechung errechnet wurde, wurde der Prozessstack mit hoher Wahrscheinlichkeit nicht durch einen anderen Prozess überschrieben. Wenn die Prüfsumme nicht dem zuvor errechneten Wert entspricht, wurden die Daten auf dem Prozessstack zwischenzeitlich von einem anderen Prozess überschrieben. Der Prozessstack ist damit inkonsistent.

Lernerfolgsfragen

- Wann tritt ein Stack Overflow auf?
- Was bedeutet *inkonsistent* im Zusammenhang mit Prozessstacks?
- Wie kann ein Stack Overflow erkannt werden?
- Können erkannte Stackinkonsistenzen behoben werden?

2.3.5 Speicheraufteilung

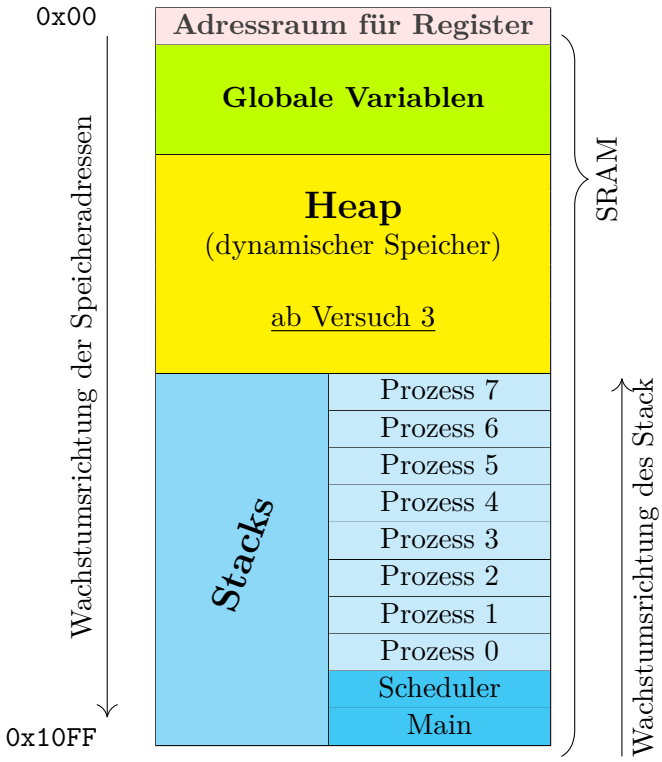

Abbildung 2.5: Speicheraufteilung

Der SRAM-Speicher des Mikrocontrollers wird in mehrere Abschnitte unterteilt, welche in Abbildung 2.5 dargestellt sind. Auf dem Speicher werden globale Variablen, ein Heap sowie die System- und Prozessstacks abgelegt. Für Stacks soll die Hälfte des SRAM verwendet werden, die andere Hälfte wird zwischen den globalen Variablen und dem Heap

aufgeteilt. Der Heap wird in Versuch 3 (*Heap / Schedulingstrategien*) behandelt, während im Folgenden die genaue Aufteilung des Speicherbereichs für die Stacks beschrieben wird: Der Stackbereich des SRAM teilt sich in den Bereich für die Systemstacks (Stack der Main-Funktion und des Schedulers) und den Bereich für die Prozessstacks auf. Der Bereich für die Prozessstacks wird gleichmäßig in so viele einzelne Prozessstacks aufgeteilt, wie die maximale Anzahl gleichzeitig laufender Prozesse vorgibt. Aus der Größe der einzelnen Speicherbereiche und der Startadresse des ersten Stacks lassen sich somit die Startadressen der übrigen Stacks berechnen.

Achtung

Der SRAM des Mikrocontrollers wird bei 0x0100 beginnend adressiert. Er teilt sich den Adressraum mit den Prozessor- und I/O-Registern.

2.3.6 Speichern von Strings

Konstante Strings nehmen sehr viel Speicher in Anspruch. Da der SRAM-Speicher des ATmega 644 bereits zu großen Teilen für den Heap und die Prozessstacks reserviert ist, ist es nicht sinnvoll, dort zusätzlich viele konstante Strings zu speichern. Es besteht die Möglichkeit, diese Daten im deutlich größeren *Flash*-Speicher des ATmega 644 abzulegen. Fügt man bei einer Variablendeklaration zum Datentyp das Compiler-Attribut `PROGMEM` hinzu, weist man den Compiler an, Daten in den Flash abzulegen.
Um Strings im Flash-Speicher abzulegen, kann ein `define` verwendet werden, welches in der Bibliothek `<avr/pgmspace.h>` vorhanden ist. Die Definition `PSTR(str)` kennzeichnet den angegebenen String `str` für den Compiler zur Ablage im Flash-Speicher. Eine solche Zeichenkette kann mit der Methode `lcd_writeProgString` auf dem LCD ausgegeben werden. Die Verwendung ist in Listing 2.1 exemplarisch gezeigt.

```
1 void example() {
2   // lcd_writeString: String im SRAM abgelegt
3   lcd_writeString("Hallo Welt");
4
5   // lcd_writeProgString: String im Flash abgelegt
6   lcd_writeProgString(PSTR("Hallo Welt"));
7 }
```

Listing 2.1: Verwendung von PSTR(str)

2.4 Hausaufgaben

Bisher wurden die Betriebssystemkonzepte lediglich allgemein behandelt. In diesem Abschnitt werden Informationen zur Implementierung dieser Konzepte in einem realen System gegeben. Zur Reduktion des Implementierungsaufwands steht für diesen Versuch ein Codegerüst zur Verfügung, in welchem bereits einige grundlegende Funktionen implementiert sind und welches Ansätze für die in diesem Versuch vorgestellten Komponenten enthält. Dieses Codegerüst soll im Rahmen der Versuchsvorbereitung als Hausaufgabe vervollständigt werden.
Lösen Sie alle hier vorgestellten Aufgaben zu Hause mithilfe von Microchip Studio 7 und schicken Sie die dabei erstellte und funktionsfähige Implementierung über Moodle ein. Ihre Abgabe soll dabei die `.atsln`-Datei, das Makefile, sowie den Unterordner mit den `.c/.h`-Dateien inklusive der `.xml/.cproj`-Dateien enthalten. Beachten Sie bei der Bearbeitung der Aufgaben die angegebenen Hinweise zur Implementierung! Ihr Code muss ohne Fehler und ohne Warnungen kompilieren.

Achtung

Verwenden Sie zur Prüfung auf Warnungen den Befehl „Rebuild Solution“ im „Build“-Menü des Microchip Studio 7. Die übrigen in der grafischen Oberfläche angezeigten Buttons führen nur ein inkrementelles Kompilieren aus, d.h. es werden nur geänderte Dateien neu kompiliert. Warnungen und Fehlermeldungen in unveränderten Dateien werden dabei nicht ausgegeben.

2.4.1 Übersicht

Dieser Versuch behandelt, wie im Grundlagenkapitel 2.1 angedeutet, die Implementierung eines Schedulers für das Betriebssystems SPOS auf dem Mikrocontroller ATmega 644 und die damit verbundenen grundlegenden Funktionalitäten und Datenstrukturen, um ein Prozess-Scheduling zu ermöglichen. Für die in Abschnitt 2.3.2 beschriebene periodische Ausführung des Schedulers wird ein sogenannter Timer-Interrupt verwendet, der regelmäßig ausgelöst wird und eine Interrupt-Service-Routine (ISR) aufruft. In dieser ISR soll die Abarbeitung eines Prozesses unterbrochen, dessen Zustand gespeichert, ein neuer Prozess ausgewählt, und dessen Abarbeitung fortgesetzt werden. Der Zustand eines Prozesses umfasst folgende Daten:

- Programm-ID: Index des Programms, das durch den Prozess ausgeführt wird (Position im Array `os_programs`)
- Prozesszustand: Der Bereitschaftsstatus des Prozesses (z. B. Ready, Running, …)
- Prozesspriorität: Die für den Scheduler relevante Priorität des Prozesses

- Stackpointer: Zeiger auf die nächste freie Speicherzelle des Prozessstacks des Prozesses
- Programmzähler: Die Adresse des aktuell ausgeführten Befehls
- Laufzeitkontext: Der aktuelle Inhalt der Prozessorregister und des Statusregisters `SREG`

Die Prozessdaten *Programm-ID*, *Prozesszustand* und *Prozesspriorität* werden in einem sogenannten Prozess-Array verwaltet. Zusätzlich befindet sich für einen wartenden Prozess dort der gespeicherte *Stackpointer*. Den *Programmzähler* und *Laufzeitkontext* eines Prozesses legen wir, sofern der Prozess auf Ausführung wartet, auf seinem Prozessstack ab. Die Funktionsweise eines Stacks wurde in Abschnitt 2.3.4 beschrieben. Läuft der Prozess gerade, befindet sich der aktuelle Programmzähler im `PC`-Register, der Laufzeitkontext in den dafür vorgesehenden anderen Prozessorregistern.
Zudem soll der Scheduler eine Implementierung kritischer Bereiche, wie im Grundlagenkapitel 2.3.3 beschrieben, umfassen. Um die Konsistenz der einzelnen Prozessstacks sicherzustellen, soll SPOS eine Heuristik zur Erkennung von Stackinkonsistenzen bereitstellen. Eine Funktion zur Fehlerbehandlung soll ebenfalls in SPOS integriert werden, auf die im folgenden Abschnitt eingegangen wird.

2.4.2 Fehlerbehandlung

Erstellen Sie als erstes eine Funktion, mit der kritische Fehler, die während der Ausführung von SPOS auftreten können, komfortabel ausgegeben werden können.

Implementierung von `os_errorPStr`

Zur Umsetzung der Fehlerausgabe ist die Funktion `os_errorPStr` in der Datei `os_core.c` zu implementieren. Sie soll die übergebene Fehlermeldung auf dem LCD ausgeben und das Betriebssystem vorläufig anhalten, indem sie die Interrupts global deaktiviert. Dies erreichen Sie, indem Sie das 7. Bit (MSB) im Statusregister (`SREG`) deaktivieren. Vor der Ausgabe des Fehlers sollte das LCD mithilfe der `lcd_clear()` Methode gelöscht werden, sodass die Ausgabe nicht von bereits existierendem Text beeinträchtigt wird. Beachten Sie zur Anzeige der Fehlermeldung auf dem Display ebenfalls die Hinweise im Kapitel 2.3.6. Mit der Tastenkombination „Enter + ESC“ soll die Fehlermeldung durch den Benutzer quittierbar sein und das Betriebssystem soll fortgesetzt werden sobald alle Tasten wieder losgelassen wurden. Es sind keine weiteren Maßnahmen bezüglich des Fehlers zu ergreifen, außer dass beim Verlassen von `os_errorPStr` die Interrupts wieder global aktiviert werden müssen, sofern Sie vor Aufruf der Funktion aktiviert waren. Fragen Sie den Status der Buttons mithilfe der Funktionen aus dem Modul `os_input` ab. Sie können das bereits in Versuch 1 (ADC-Menü) implementierte `os_input`-Modul wiederverwenden. Es ist durchaus möglich, dass sich SPOS nach dem manuellen Quittieren eines Fehlers nicht mehr korrekt verhält. Aus diesem Grund sollte die Möglichkeit der manuellen Quittierung nur zu Debuggingzwecken genutzt werden.

Verwendung von `os_error`

An einigen Stellen Ihrer Implementierung ist es sinnvoll, bestimmte Fehler abzufangen und dem Benutzer auf dem Display anzuzeigen. In der Regel sollen keine Fehler abgefangen werden, die behandelt werden können. Nutzen Sie hierzu, wie in Abbildung 2.6 beispielhaft dargestellt, das im Codegerüst bereits vorgegebene Makro `os_error`.

```
os_error("Hallo Welt");
```

Abbildung 2.6: Verwendung von `os_error`

2.4.3 Verwaltung von Programmen und Prozessen

Implementieren Sie die im Folgenden erläuterten Funktionen, Datentypen und Variablen zur Verwaltung von Programmen und Prozessen. Zur Bearbeitung ist es notwendig, das Grundlagenkapitel gelesen zu haben.

Hinweise zur Implementierung

Damit das Betriebssystem mit Programmen und Prozessen arbeiten kann, ist es nötig, zu jedem Prozess Verwaltungsinformationen zu speichern. Diese werden in einer speziell für diesen Zweck entworfenen Datenstruktur `Process` gespeichert, die von Ihnen angelegt werden muss. In den folgenden Abschnitten werden die Komponenten dieser Datenstruktur erläutert.

Hinweis

Aufgrund der Interaktion mit dem Taskmanager müssen für die Attribute dieses Datentyps bestimmte Namen gewählt werden. Abweichungen von den vorgegebenen Bezeichnern führen zur Fehlfunktion des Taskmanagers.

Programme und Prozesse Programme werden in SPOS als parameterlose Funktionen ohne Rückgabewert definiert.

1. Legen Sie in `os_process.h` die Typbenennung `typedef void Program(void)` an.
2. Legen Sie das Array `os_programs` von Zeigern auf „`Program`“ in der Datei `os_scheduler.c` an. Es soll bei Ausführung genau einen Funktionszeiger auf jedes registrierte Programm enthalten. Die maximale Anzahl Programme wird mit dem `define MAX_NUMBER_OF_PROGRAMS` in der Datei `defines.h` festgelegt.

Prozesse sind als Instanzen von Programmen zu verstehen.

1. Implementieren Sie in der Datei `os_process.h` die Struktur `struct Process`, die wie in Abschnitt 2.4.1 angedeutet, die Prozessdaten *Programm-ID*, *Prozesszustand*, *Prozesspriorität* und *Stackpointer* zusammenfasst. Verwenden Sie als Datentyp für den Prozesszustand `state` das `enum ProcessState` und für den Stackpointer `sp` das `union Stackpointer`. Der Wert des Stackpointers sollte als `uint16_t` abrufbar sein, es sollte aber auch der direkte Zugriff auf den referenzierten Speicher möglich sein. Ein union ermöglicht dies ohne die Notwendigkeit den Wert regelmäßig casten zu müssen. Als Datentyp für die Programm-ID `progID` und Prozesspriorität `priority` verwenden Sie die teilweise noch von Ihnen anzulegenden Typbenennungen `ProgramID` und `Priority` für den Typ `uint8_t`. Die Struktur wurde bereits mittels `typedef` als `Process` benannt, um diese später lesbarer und komfortabler verwenden zu können.

2. Um dem Scheduler den Zugriff auf alle Prozesse zu ermöglichen, muss das Prozess-Array `os_processes` in der Datei `os_scheduler.c` global angelegt werden. Dieses Array soll genau `MAX_NUMBER_OF_PROCESSES` Einträge des Typs `Process` enthalten. Die Einträge dieses Arrays werden als *Prozessslots* bezeichnet.

Hinweis

Benutzen Sie zur Implementierung der Funktion `os_exec` an geeigneten Stellen die Defines `PROCESS_STACK_BOTTOM(PID)` und gegebenenfalls `STACK_SIZE_PROC` aus der Datei `defines.h`, wie in Abbildung 2.7 dargestellt.

Starten von Programmen Zum Start eines Programms wird ein neuer Prozess als Instanz dieses Programms erzeugt und die relevanten Prozessinformationen gespeichert. Dazu wird die Funktion `os_exec` verwendet, welche im Codegerüst angelegt ist und ergänzt werden muss. Die Funktion hat die Aufgabe, im nächsten freien Slot des Arrays `os_processes` die notwendigen Informationen über den neu zu erzeugenden Prozess abzulegen und den benötigten Stackspeicher der Prozesse vorzubereiten. Dazu führt `os_exec` zunächst die folgenden Schritte durch:

1. Freien Platz im Array `os_processes` finden

2. Den zu dem angegebenen Programmindex gehörigen Funktionszeiger aus dem Array `os_programs` laden und überprüfen

3. Programmindex, Prozesszustand und Prozesspriorität speichern

4. Prozessstack vorbereiten

Im Folgenden werden die einzelnen Schritte erläutert. Sobald alle diese Schritte durchgeführt wurden, steht der neue Prozess dem Scheduler zur Verfügung.

1. Freien Platz im Array `os_processes` *finden:* Der erste freie Platz in `os_processes` soll für den neu zu erzeugenden Prozess genutzt werden. Sind alle Plätze belegt, soll `os_exec` den Rückgabewert `INVALID_PROCESS` zurückgeben. Ob ein Platz bereits belegt ist, lässt sich anhand des Prozesszustands erkennen.

2. Funktionszeiger aus dem Array `os_programs` *laden:* Zum Laden des Funktionszeigers soll die Funktion `Program* os_lookupProgramFunction(ProgramID programID)` verwendet werden, die im Fehlerfall `NULL` zurückgibt. Ein denkbarer Fehlerfall wäre z. B. die Übergabe eines ungültigen Programmindex. `NULL` ist bereits in der `defines.h` definiert und kann direkt verwendet werden. Wurde der Wert `NULL` zurückgegeben, soll die Ausführung der Funktion mit `INVALID_PROCESS` als Rückgabewert abgebrochen werden.

3. Programmindex, Prozesszustand und Prozesspriorität speichern: Der Prozesszustand gibt den aktuellen Status eines Prozesses an. Wenn ein neuer Prozess erzeugt wird, soll dieser initial den Zustand `OS_PS_READY` erhalten.
Ebenso wie der Prozesszustand muss auch der Programmindex gespeichert werden, da es möglich sein soll, dem laufenden Prozess das von ihm ausgeführte Programm zuzuordnen.
Zusätzlich muss die Prozesspriorität gespeichert werden, um priorisierende Scheduling-Strategien anwenden zu können.

4. Prozessstack vorbereiten: Wie in den Grundlagen im *Abschnitt Prozessstacks* erläutert, liegen Rücksprungadressen auf dem Stack. Das Starten eines Prozesses wird wie der Rücksprung aus einer Unterfunktion behandelt. Dabei wird eine 16 Bit breite Programmadresse vom Stack in das `PC`-Register geladen. Um einen Prozess zu starten, müssen daher entsprechende initiale Daten im Voraus auf dessen Prozessstack gespeichert werden. Diese Daten umfassen die initiale Rücksprungadresse, und, da wir bei unterbrochenen Prozessen den Laufzeitkontext auf dem Stack zwischenspeichern, den initialen Laufzeitkontext. Verwenden Sie zum Schreiben auf den Stack das `union StackPointer`.
Als Programmzähler wird für den Start eines Prozesses der zu dessen Programm im Array `os_programs` hinterlegte Funktionszeiger verwendet. Das Low-Byte der insgesamt 16 Bit breiten Adresse muss hierbei als erstes auf dem Stack gesichert werden. An der nächstkleineren Stelle wird das High-Byte geschrieben.
Der Laufzeitkontext besteht aus den 32 Prozessorregistern und dem Statusregister `SREG`, die alle mit dem Byte `0` initialisiert werden müssen. Daher werden 33 Null-Bytes nach dem Funktionszeiger auf den Prozessstack geschrieben. Anschließend muss der Stackpointer des neuen Prozesses auf die erste freie Speicherstelle des Prozessstacks gesetzt werden. Verwenden Sie hierzu das `define PROCESS_STACK_BOTTOM(PID)`, welches die erste (höchste) Speicherstelle des Prozessstacks mit ID *PID* bereitstellt.

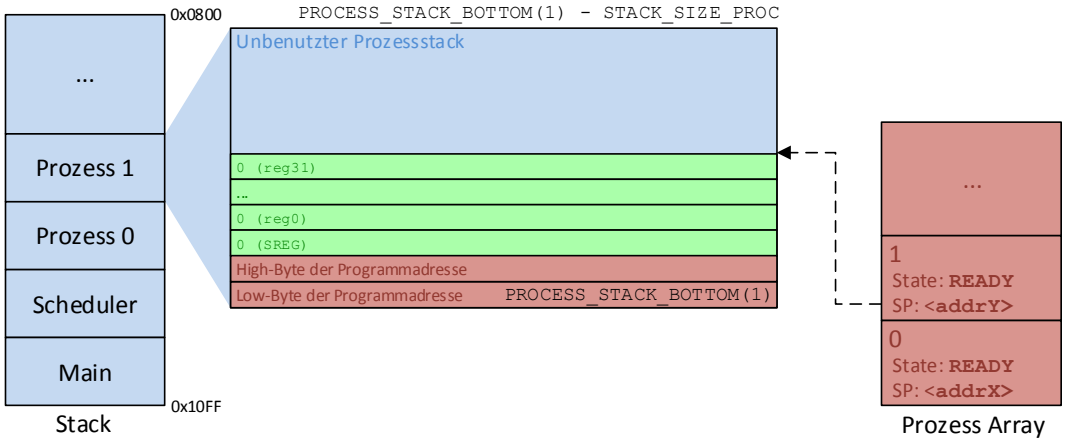

Abbildung 2.7: Prozesstack 1 nach Ausführung von `os_exec`

Abbildung 2.7 stellt beispielhaft den Prozessstack nach Ausführung von `os_exec` dar, wobei Prozessslot 1 als erster freier Platz im Prozess-Array `os_processes` gefunden wurde. Bitte beachten Sie, dass `os_exec` keine Fehlermeldungen auf dem Display ausgeben soll.

Überprüfung Ihrer Implementierung

Sie können den Testtask `os_exec` benutzten, um Ihre Implementierung der Funktion `os_exec` zu überprüfen.

2.4.4 Implementierung des Schedulers

Dieser Abschnitt behandelt die Implementierung der Scheduler-ISR, die regelmäßig durch einen im Codegerüst vorkonfigurierten Timer aufgerufen wird, um dem Benutzer den Eindruck zu vermitteln, dass Prozesse gleichzeitig ausgeführt werden. In dieser ISR soll die Abarbeitung eines Prozesses unterbrochen, dessen Zustand gespeichert, ein neuer Prozess ausgewählt und dessen Abarbeitung fortgesetzt werden.

Scheduler initialisieren

Die im Codegerüst vorhandene Funktion `os_initScheduler` soll beim Start des Betriebssystems das Prozess-Array so initialisieren, dass für jedes Programm, das mit `AUTOSTART` markiert ist, ein Prozess instanziiert wird.
Bereiten Sie das Array `os_processes` vor, indem Sie das Feld `state` aller Einträge auf `OS_PS_UNUSED` setzen. Durchlaufen Sie anschließend das Array `os_programs`, welches vor Ausführung von SPOS durch das im Codegerüst bereitgestellte Makro `PROGRAM` automatisch mit entsprechenden Programmen initialisiert wurde. Verwenden Sie die Funktion `os_checkAutostartProgram(ProgramID prog)`, um zu überprüfen, ob das Programm

mit Programmindex `prog` automatisch gestartet werden soll. Ist ein Programm zum automatischen Starten markiert, soll es an dieser Stelle mit der Funktion `os_exec` und Priorität `DEFAULT_PRIORITY` gestartet werden.

Überprüfung Ihrer Implementierung Sie können den Testtask `os_initScheduler` benutzten, um Ihre Implementierung der Funktion `os_initScheduler` zu überprüfen.

Hinweise zum Debuggen

Ein falsch initialisierter oder wiederhergestellter Stack hat höchtwahrscheinlich zur Folge, dass der Mikrocontroller abstürzt. Beispielsweise kann eine inkorrekte Rücksprungadresse den Sprung an eine Stelle des Programmspeichers, an der kein passender Code steht, verursachen. Der Prozessor befindet sich dann in einem Zustand, in dem keine Programmausführung mehr möglich ist und der Mikrocontroller startet sich neu.
Die im Codegerüst vorgegebene Funktion `os_init` zeigt den Grund für den Neustart an. Im beschriebenen Fall des Software-Resets wird auf dem LCD „Soft-Reset – System Error“ ausgegeben. Die Meldung muss daraufhin quittiert werden.
Andere Gründe für einen Reset werden im *MCU Status Register* hinterlegt. Im Codegerüst wird dieses ausgelesen und ein Kürzel angezeigt, der den Resetgrund angibt. Die für Sie relevanten Gründe seien folgend aufgeführt:

JT - JTag Reset Der JTag veranlasste den Neustart, beispielsweise durch Flashen von neuem Code.

EXT - External Reset Der Neustart wurde durch eine logische 0 am Reset-Pin veranlasst, beispielsweise durch Drücken des Reset-Knopfes auf dem Evaluationsboard.

POW - Power Reset, BO - Brownout Detection Reset Bei einer Anzeige von POW oder BO ist eine fehlerhafte Stromversorgung des Mikrocontrollers Grund des Neustartes.

Speichern des aktuellen Prozesses

Der derzeit laufende Prozess muss jederzeit durch einen Aufruf der Funktion `os_getCurrentProc` abfragbar sein. Legen Sie dazu an geeigneter Stelle eine globale Variable mit dem Namen `currentProc` an und implementieren Sie die Funktion `os_getCurrentProc` mit geeignetem Rückgabewert.

Implementierung der Scheduler-ISR

Die im Codegerüst vorhandene Scheduler-ISR `TIMER2_COMPA_vect` wird durch einen Timer automatisch in einem Intervall von $\approx 3\,\text{ms}$ aufgerufen. Erweitern Sie sie um die Fähigkeit, zwischen verschiedenen Prozessen zu wechseln.
Wird ein Prozess unterbrochen, muss sein Laufzeitkontext auf dem Stack gesichert werden. Ebenso muss der aktuelle Stackpointer durch den Scheduler gespeichert werden,

damit der Prozess später an derselben Stelle fortgesetzt wird. Beachten Sie dabei, dass die ISR als `naked` deklariert ist. Wenn eine Funktion als `naked` deklariert wird, bedeutet das für den Compiler, dass für den Aufruf der Funktion keine Prozessregister gesichert werden sollen und kein Speicher für lokale Variablen angelegt werden soll. Daher sollten keine lokalen Variablen in der ISR selber verwendet werden, denn dies kann zu undefiniertem Verhalten führen.

Hinweis

Benutzen Sie zum Setzen des Stackpointers auf den Scheduler-Stack das Define `BOTTOM_OF_ISR_STACK`.

Wechsel zwischen Prozessstacks Der vom Prozessor verwendete aktuelle Stackpointer ist im Register `SP`[1] gespeichert. Das Register `SP` enthält die Adresse des ersten freien Speicherbereichs des aktuell verwendeten Prozessstacks. Um zwischen zwei Prozessstacks zu wechseln, wird zunächst der Inhalt des Registers `SP` als Stackpointer des aktuellen Prozesses gespeichert. Hierfür existiert bereits ein `union`, das als `StackPointer` benannt wurde und im Codegerüst vorgegeben ist. Anschließend wird das Register `SP` auf den Stackpointer des nächsten Prozesses gesetzt. Nachfolgende Stackoperationen beziehen sich anschließend auf den Prozessstack des nächsten Prozesses.

Ablauf des Schedulers und Prozesswechsel Der Ablauf des Schedulers und der damit durchgeführte Prozesswechsel umfasst folgende Schritte:

1. Speichern des Programmzählers als Rücksprungadresse für die spätere Fortsetzung des aktuellen Prozesses auf dessen Prozessstack. Dies wird implizit beim Sprung in die ISR erledigt
2. Sichern des Laufzeitkontextes des aktuellen Prozesses auf dessen Prozessstack. Dies erfolgt durch das Assemblermakro `saveContext()`
3. Sichern des Stackpointers für den Prozessstack des aktuellen Prozesses
4. Setzen des `SP`-Registers auf den Scheduler-Stack
5. Setzen des Prozesszustandes des aktuellen Prozesses auf `OS_PS_READY`
6. Auswahl des nächsten fortzusetzenden Prozesses durch Aufruf der aktuell verwendeten Schedulingstrategie

[1] Auf das `SP`-Register kann wie auf beispielsweise auch das `DDRA`-Register direkt aus dem Programmcode zugegriffen werden. Sollte Microchip Studio 7hier durch rot unterstrichenen Code Fehler anzeigen, können Sie dies ignorieren; kompilieren funktioniert trotz diesem Hinweis.

7. Setzen des Prozesszustandes des fortzusetzenden Prozesses auf `OS_PS_RUNNING`
8. Wiederherstellen des Stackpointers für den Prozessstack des fortzusetzenden Prozesses
9. Wiederherstellen des Laufzeitkontextes des fortzusetzenden Prozesses durch das Assemblermakro `restoreContext()`
10. Automatischer Rücksprung durch das Assemblermakro `restoreContext()`

Abbildung 2.8 illustriert den Zustand von SPOS vor dem Prozesswechsel, wobei der Stack auf der rechten Seite, Register des Mikrocontrollers oben und Datenstrukturen zur Prozessverwaltung links angeordnet sind. Abbildungen 2.9 und 2.10 illustrieren Schritte 1 bis 3 bzw. 8 bis 10 des Schedulers anhand eines beispielhaften Prozesswechsels von Prozess 1 nach Prozess 2. Die Position des Stackpointers ist durch eine gestrichelte, Wertzuweisungen durch eine durchgezogene Linie dargestellt. Beachten Sie, dass nach jeder Stackoperation `push` oder `pop` der Stackpointer de- bzw. inkrementiert wird und Abbildung 2.9 den Stackpointer nach Schritt 2 bzw. 2.10 vor Schritt 9 andeutet. Das Zusatzdokument *Schedulerablauf*, welches Sie im ł2p finden, stellt den Schedulerablauf schrittweise dar, wobei die Position des Stackpointers nach jedem Schritt aktualisiert wurde.

Wenn diese Operationen ausgeführt worden sind, steht dem fortgesetzten Prozess der Laufzeitkontext zur Verfügung, den dieser vor der Unterbrechung durch den Scheduler hatte. Abbildung 2.11 illustriert dies anhand des obigen Beispiels. Die Ausführung wird an der Stelle fortgesetzt, an der sie zuvor durch den Scheduler unterbrochen wurde, da die zuvor auf dem Stack abgelegten Register und der Programmzähler wiederhergestellt worden sind. Der Kontext der ISR (und somit der Wert des Stackpointers am Ende der ISR) muss nicht gespeichert werden, da eine ISR nicht erwartet, auf dem Stack Daten zu finden. Es ist daher vollkommen richtig, den Stackpointer bei jedem Aufruf der ISR wieder auf den Beginn des Scheduler-Stacks zu setzen.

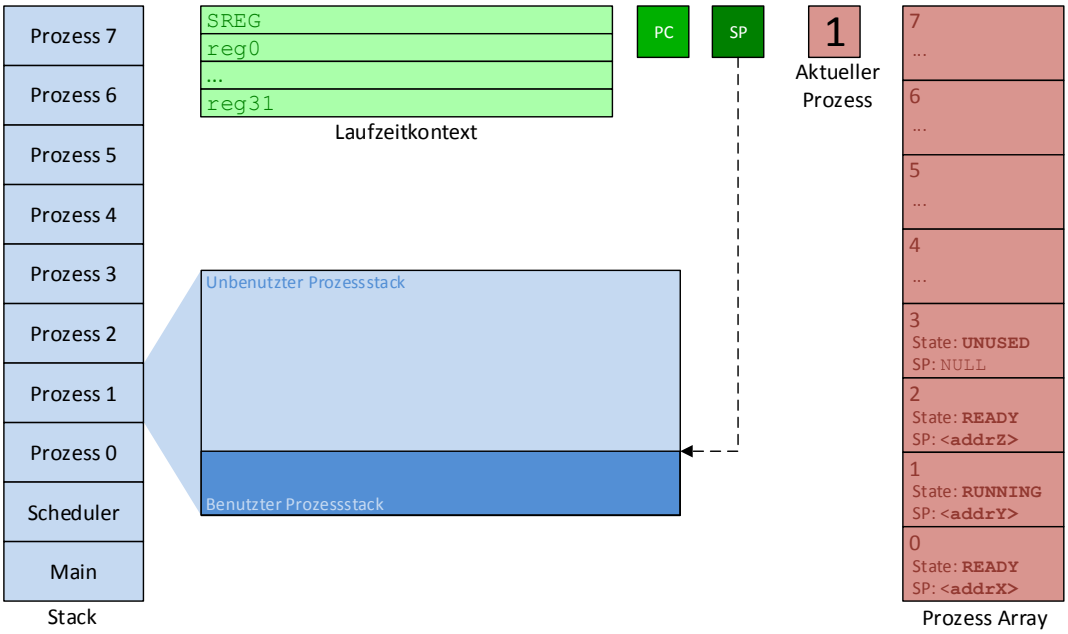

Abbildung 2.8: Zustand vor Prozesswechsel von Prozess 1 nach 2

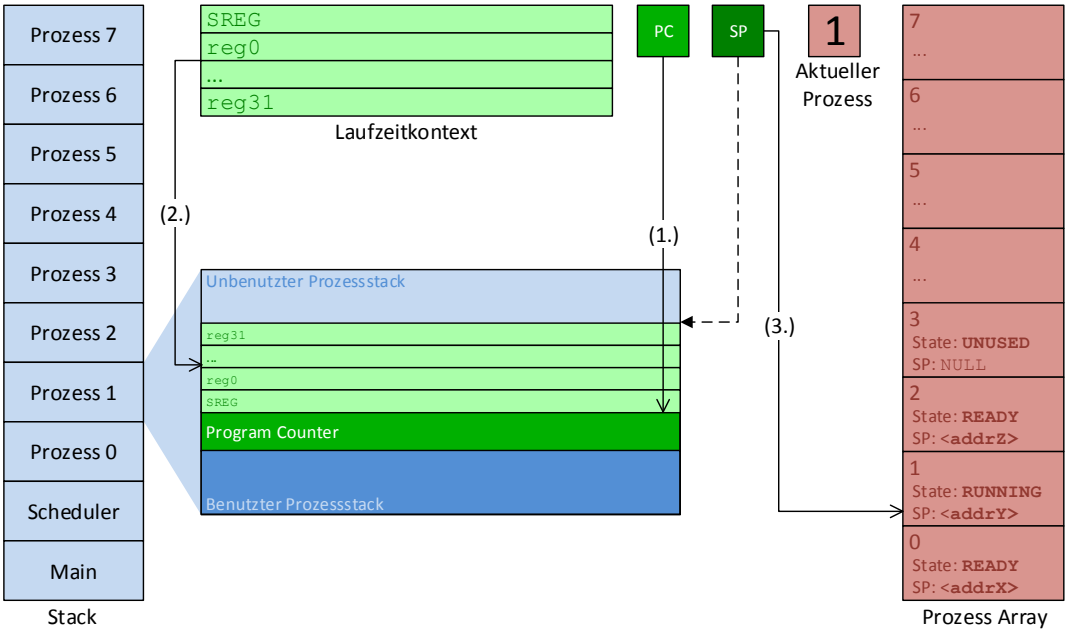

Abbildung 2.9: Schritte 1-3 des Prozesswechsels von Prozess 1 nach 2

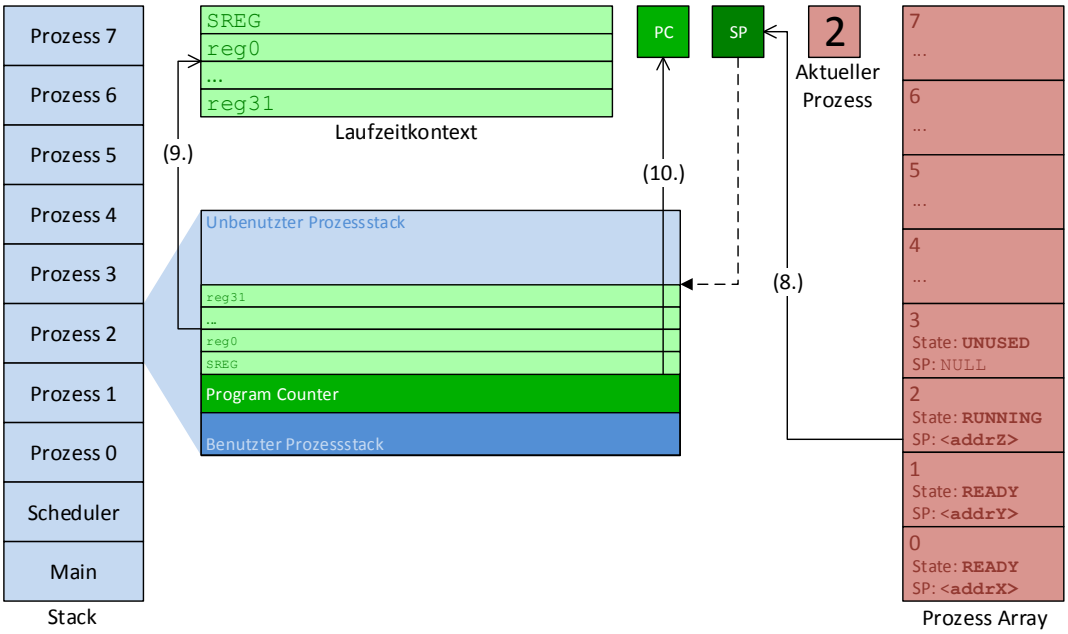

Abbildung 2.10: Schritte 8-10 des Prozesswechsels von Prozess 1 nach 2

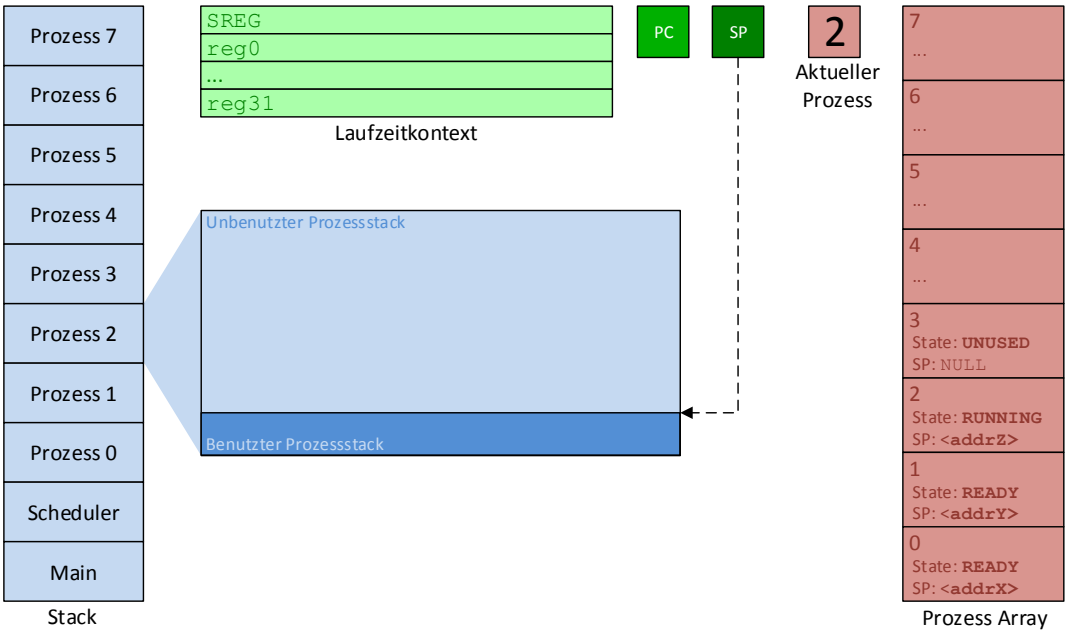

Abbildung 2.11: Zustand nach Prozesswechsel von Prozess 1 nach 2

Implementierung des Leerlaufprozesses

Erweitern Sie in der Datei `os_scheduler.c` das Programm `PROGRAM(0, AUTOSTART)`, sodass dieses in einer Endlosschleife einen Punkt „.“auf dem LCD ausgibt und so viele Millisekunden wartet, wie das `define DEFAULT_OUTPUT_DELAY` angibt. Sie können die Funktion `delayMs(int ms)` benutzen, um eine entsprechende Verzögerung zu realisieren. Dieses Programm wird *Leerlaufprogramm*, seine laufende Instanz *Leerlaufprozess* genannt. Der Leerlaufprozess ist der erste Prozess, dem Rechenzeit zugeteilt werden soll. Sorgen Sie daher dafür, dass dieser in der Funktion `os_startScheduler` als erstes gestartet wird. Nach dem ersten Prozesswechsel soll der Leerlaufprozess nur noch aufgerufen werden, sofern kein anderer Prozess zur Verfügung steht.

Starten des Schedulers

Implementieren Sie die Funktion `os_startScheduler`. Die durchzuführenden Schritte sind:

1. Setzen der Variable für den aktuellen Prozess auf `0` (Leerlaufprozess)
2. Den Zustand des Leerlaufprozesses auf `OS_PS_RUNNING` ändern
3. Setzen des Stackpointers auf den Prozessstack des Leerlaufprozesses
4. Sprung in den Leerlaufprozess mit `restoreContext()`

Schedulingstrategien

Implementieren Sie die Schedulingstrategien *Random* und *Even*. Benutzen Sie die Funktion `rand()`, um eine Zufallszahl zwischen 0 und 32767 zu erzeugen. Dabei soll darauf geachtet werden, dass die Funktion `rand()` nur ein einziges Mal aufgerufen wird und dass die Auswahl der Prozesse gleichverteilt ist. Gehen Sie daher so vor, dass Ihre Implementation bei einem Rückgabewert der `rand()` Funktion von 0 denjenigen Prozess zurück gibt, welcher von allen Prozessen im `OS_PS_READY` Zustand die kleinsten Prozess ID besitzt. Ist der Rückgabewert 1, so soll der Prozess mit der zweitkleinsten ID zurückgeben werden usw. Falls der Rückgabewert höher ist als die Anzahl an Prozessen, welche sich im `OS_PS_READY` Zustand befinden, so fangen Sie wieder von vorne an. Um dieses Verhalten zu erreichen kann der Modulo-Operator (`%`) genutzt werden.
Um die Auswahl zwischen mehreren Schedulingstrategien zu ermöglichen, soll eine globale Variable angelegt werden, welche die aktuell verwendete Schedulingstrategie speichert. Lese- und Schreibvorgänge sollen über die Funktionen `os_getSchedulingStrategy` und `os_setSchedulingStrategy` geschehen. Diese Funktionen sind im Codegerüst bereits vorhanden und müssen ergänzt werden. Verwenden Sie zur Speicherung der aktuellen Schedulingstrategie den `enum`-Typ `SchedulingStrategy`. Als mögliche Strategien werden in diesem Versuch die Strategien *Random* und *Even* betrachtet. Die Implementierung der Strategien *RunToCompletion*, *RoundRobin* und *InactiveAging* ist eine optionale Präsenzaufgabe. Alle Schedulingstrategien sollen ausschließlich Prozesse auswählen, welche

sich im Zustand `OS_PS_READY` befinden. Der Leerlaufprozess soll bei allen Strategien nur dann ausgewählt werden, wenn kein anderer Prozess verfügbar ist.

2.4.5 Implementierung kritischer Bereiche

Implementieren Sie die im Codegerüst enthaltenen Funktionen zur Verwendung kritischer Bereiche. In Abschnitt 2.3.3 wurde das Prinzip von kritischen Bereichen konzeptionell eingeführt. In SPOS wird dieses Prinzip benötigt, um das konfliktfreie Nutzen von Betriebsmitteln zu gewährleisten. Das Betreten eines kritischen Bereichs durch einen Prozess soll den Prozess exklusiv ausführen, ohne durch den Scheduler unterbrochen zu werden. Dazu wird der Scheduler während der Verarbeitung eines kritischen Bereiches deaktiviert.

Hinweise zur Implementierung

Um kritische Bereiche zu verwalten, sollen die Funktionen `os_enterCriticalSection` und `os_leaveCriticalSection` in `os_scheduler.c` implementiert werden. Diese Funktionen sollen einen kritischen Bereich betreten, bzw. verlassen, indem sie den Scheduler aus- bzw. wieder einschalten. Diese Funktionen müssen atomar sein, dürfen also nicht durch Interrupts unterbrochen werden.
Um den Scheduler in der Funktion `os_enterCriticalSection` zu deaktivieren, muss das Bit `OCIE2A` im Register `TIMSK2` auf den Wert `0` gesetzt werden. Das Löschen des Bits sorgt dafür, dass es nicht zu einem Timer-Interrupt, und somit zu einem Scheduleraufruf, kommt.
Da in Unterfunktionen weitere kritische Bereiche aufgerufen werden können, muss es möglich sein, kritische Sektionen ineinander zu verschachteln. Legen Sie dazu die globale Variable `criticalSectionCount` an, um die Verschachtelungstiefe speichern zu können. Der Ablauf beim Betreten von `os_enterCriticalSection` sollte sich an den folgenden Schritten orientieren:

1. Speichern des Global Interrupt Enable Bit (`GIEB`) aus dem `SREG`[2] in einer lokalen Variable
2. Deaktivieren des Global Interrupt Enable Bit, um das atomare Verhalten der Funktion zu gewährleisten
3. Inkrementieren der Verschachtelungstiefe des kritischen Bereiches
4. Deaktivieren des Schedulers
5. Wiederherstellen des (zuvor gespeicherten) Zustandes des Global Interrupt Enable Bit im `SREG`

[2] Hinweise zum SREG-Register finden Sie im Datenblatt des Mikrocontrollers ATmega 644 in Kapitel 5.4

Die Funktion `os_leaveCriticalSection` soll analog zu `os_enterCriticalSection` implementiert werden, mit dem Unterschied, dass der Scheduler nur dann wieder aktiviert wird, wenn nach dem Dekrementieren der Verschachtelungstiefe diese den Wert `0` angenommen hat. Somit wird der Scheduler erst nach dem Verlassen aller ineinander verschachtelten kritischen Bereiche wieder aktiviert. Führen Sie an geeigneten Stellen innerhalb dieser Funktion Fehlerüberprüfungen ein. Beispielsweise ist es ein Fehler, wenn `os_leaveCriticalSection` häufiger aufgerufen wurde, als `os_enterCriticalSection`.

Integration von kritischen Sektionen Erweitern Sie nach der Implementierung der oben beschriebenen Funktionalität die von Ihnen implementierte Funktion `os_exec` um eine kritische Sektion, welche die bisherige Implementierung vor Unterbrechung durch den Scheduler absichert.

Überprüfung Ihrer Implementierung

Zu diesem Zeitpunkt sind die grundlegenden Komponenten des Betriebssystems funktionsfähig. Sie können die Testtasks *Critical*, *Init*, *Multiple*, *Resume* und *Error* benutzen, um Ihre bisherige Arbeit zu überprüfen. Erläuterungen zu den jeweiligen Testtasks finden Sie in der im ł2p verfügbaren Testtaskdokumentation.

2.4.6 Erkennung von Stackinkonsistenzen

Erweitern Sie die zuvor erstellte Scheduler-ISR `TIMER2_COMPA_vect` um die Fähigkeit, Stackinkonsistenzen zu erkennen. Vervollständigen Sie dazu die Funktion `StackChecksum os_getStackChecksum` in `os_scheduler.c`.

Implementierung von `os_getStackChecksum`

Es soll eine Heuristik zur Erkennung von Inkonsistenzen der Prozessstacks implementiert werden. Diese Heuristik soll mit einer Funktion arbeiten, welche die Prüfsumme eines Prozessstacks berechnet. Zur Berechnung der Prüfsumme sollen alle Bytes des Prozessstacks mit dem bitweisen XOR verknüpft werden. Diese Prüffunktion hat den Vorteil, dass sie mit wenig Aufwand zu berechnen ist und Aufrufe dieser Funktion daher in relativ kurzer Zeit abgearbeitet werden können. Der Typ `StackChecksum` ist bereits mit geeignetem Wertebereich vorgegeben. Es genügt, die Daten in die Überprüfung einzubeziehen, die zwischen dem Anfang des Prozessstacks und dem Stackpointer des entsprechenden Prozesses liegen. Daten, die jenseits dieses Stackpointers liegen, werden vom Prozess nicht genutzt und sind für die Konsistenz irrelevant.

Integration in das Betriebssystem

Erweitern Sie die Struktur `struct Process` um das Attribut `checksum` mit Datentyp `StackChecksum`, welches eine Prüfsumme des Prozessstacks speichert. Beachten Sie beim Start von Prozessen durch die Funktion `os_exec`, dass diese Prüfsumme nach der Vorbereitung des Prozessstacks korrekt initialisiert werden muss. Erweitern Sie nun den

Scheduler so, dass dieser nach dem Setzen des Stackpointers auf den Scheduler-Stack die Prüfsumme des Prozessstacks des unterbrochenen Prozesses ermittelt und abspeichert. Ein im Scheduler gewählter Prozess soll nur fortgesetzt werden, sofern sich die Prüfsumme seit dem Unterbrechen nicht geändert hat. Sollte er sich geändert haben, genügt es mit `os_error` eine Fehlermeldung auszugeben.

Überprüfung Ihrer Implementierung

Sie können den Testtask *Stack Consistency* benutzen, um Ihre Erkennung von Stackinkonsistenzen zu überprüfen. Erläuterungen zu diesem Testtask finden Sie in der im ł2p verfügbaren Testtaskdokumentation.

2.4.7 Aufruf des Taskmanagers

Ein Taskmanager für SPOS wurde Ihnen bereits mit den Dateien `os_taskman.c /.h`, sowie `os_user_privileges.c /.h` vorgegeben. Der Taskmanager soll aus dem Scheduler heraus aufgerufen werden, wenn „Enter + ESC “des Evaluationsboards gleichzeitig gedrückt werden. Im Taskmanager werden verschiedene Möglichkeiten zur Interaktion mit dem Betriebssystem zur Verfügung gestellt. Es ist beispielsweise möglich, weitere Programme zu starten oder die Schedulingstrategie zu wechseln. Ergänzen Sie den Aufruf des Taskmanagers in Ihrer Implementierung des Schedulers.

Hinweise zur Implementierung Damit der Taskmanager nicht beliebig, sondern immer aus einem definierten Zustand heraus aufgerufen wird, sollen die Buttons in der Scheduler-ISR geprüft werden. Es ist sinnvoll, die Überprüfung der Buttons nach Schritt vier und vor Schritt fünf des in Abschnitt 2.4.4 vorgestellten Schedulerablaufs vorzunehmen.
Beachten Sie, dass nach dem Drücken der Buttons gewartet werden muss, bis diese alle wieder losgelassen wurden. Ansonsten würde der startende Taskmanager die noch gedrückten Buttons sofort wieder als Eingabe auffassen, was verhindert werden soll.

Lernerfolgsfragen

- Worin besteht der Unterschied zwischen atomaren und kritischen Bereichen?
- Wie werden kritische bzw. atomare Bereiche im Code erzeugt?
- Wie legen Sie ein neues Programm in SPOS an?

Hinweis

Der Taskmanager enthält Funktionen, die in diesem Versuch noch nicht verfügbar sind. Er kann mit dem `define VERSUCH` in der Datei `defines.h` an den jeweiligen Versuch angepasst werden.

2.4.8 Zusammenfassung

Folgende Übersicht listet alle Typen, Funktionen und Aufgaben auf. Alle aufgelisteten Punkte müssen zur Teilnahme am Versuch bis zur Abgabefrist bearbeitet und hochgeladen werden. Diese Übersicht kann als Checkliste verwendet werden und ist daher mit Checkboxen versehen.

- ☐ `os_core:`
  - ☐ `void os_errorPStr(char const* str)`
- ☐ `os_input:`
  - ☐ `void os_initInput()`
  - ☐ `uint8_t os_getInput()`
  - ☐ `void os_waitForInput()`
  - ☐ `void os_waitForNoInput()`
- ☐ `os_process:`
  - ☐ `struct Process`
  - ☐ `Process`, als Typbenennung für `struct Process`
  - ☐ `ProcessID`, als Typbenennung für `uint8_t`
  - ☐ `ProgramID`, als Typbenennung für `uint8_t`
  - ☐ `Program`, als Typbenennung für `void(void)`
    (Vgl. „Begleitendes Dokument für das Praktikum Systemprogrammierung“ Abschnitt Funktionspointer)
- ☐ `os_scheduler:`
  - ☐ `Program* os_programs[MAX_NUMBER_OF_PROGRAMS]`
  - ☐ `Process os_processes[MAX_NUMBER_OF_PROCESSES]`
  - ☐ `uint8_t criticalSectionCount`
  - ☐ `void os_initScheduler()`
  - ☐ `void os_startScheduler()`
  - ☐ `StackChecksum os_getStackChecksum(ProcessID pid)`
  - ☐ `ProcessID os_exec(ProgramID programID, Priority priority)` (inkl. kritische Sektionen)
  - ☐ `ProcessID currentProc`
  - ☐ `ProcessID os_getCurrentProc()`
  - ☐ `ISR (TIMER2_COMPA_vect)`
    (Scheduler-ISR)
  - ☐ `void os_enterCriticalSection()`

- `void os_leaveCriticalSection()`
- `void os_setSchedulingStrategy(SchedulingStrategy strategy)`
- `SchedulingStrategy os_getSchedulingStrategy(void)`
- `PROGRAM(0, AUTOSTART)` (Leerlaufprozess)

`os_scheduling_strategies:`

- `ProcessID os_Scheduler_Random(Process const processes[], ProcessID current)`
- `ProcessID os_Scheduler_Even(Process const processes[], ProcessID current)`

2.5 Präsenzaufgaben

Die folgenden Aufgaben müssen bis zum Ende des Praktikumstermins umgesetzt werden. Es empfiehlt sich, die Aufgaben schon vor dem Praktikumstermin zu bearbeiten und nach Möglichkeit fertigzustellen. Es ist erlaubt, die Praktikumstermine nach Abschluss aller Präsenzaufgaben und Abnahme dieser durch einen Betreuer früher zu verlassen.

2.5.1 Testtasks

Sie finden im Moodle eine Sammlung von Testtasks, mit denen die Funktionalität Ihres Betriebssystems zum aktuellen Entwicklungsstand getestet werden kann. Am Ende des Versuchs müssen alle nicht optionalen Testtasks fehlerfrei laufen. Der korrekte Ablauf der Testtasks gemäß der ebenfalls im Moodle verfügbaren Beschreibung wird während des Versuchs geprüft. Die Implementierungshinweise, Achtung-Boxen und Lernerfolgsfragen in diesen Unterlagen weisen meist auf notwendige Kriterien für den erfolgreichen Durchlauf der Testtasks hin.
Wenn Ihre selbst entwickelten Anwendungsprogramme fehlerfrei unterstützt werden, starten Sie die Testtasks. Wenn es mit diesen Probleme gibt, haben Sie wahrscheinlich bestimmte Anforderungen nicht erfüllt, oder gewisse Sonderfälle nicht bedacht. Diese Sonderfälle können bei der Erweiterung Ihrer Implementierung des Betriebssystems für spätere Versuche zu Folgefehlern führen. Ergänzen oder korrigieren Sie Ihr Projekt, wenn Probleme mit den Testtasks auftreten.

Achtung

Sollten Sie alle zur Verfügung gestellten Testtasks erfolgreich ausführen können, ist dies keine Sicherheit dafür, dass Ihr Code fehlerfrei ist. Diese Testtasks decken lediglich einen Teil der möglichen Fehler ab.

2.5.2 Implementierung der Schedulingstrategien (in Versuch 2 optional)

Sollten Sie alle anderen Aufgaben vor Ende des Praktikumstermins abgearbeitet haben, implementieren Sie die Schedulingstrategien *RunToCompletion*, *RoundRobin* und *InactiveAging* und die Initialisierung dieser wie in Kapitel 2.3.2 beschrieben. Sie dürfen den Praktikumsraum erst verlassen, wenn alle optionalen Aufgaben korrekt bearbeitet wurden oder die Zeit abgelaufen ist. Beachten Sie die Hinweise zur Initialisierung der Schedulingstrategien in Abschnitt 2.5.2.

Initialisierung der Schedulingstrategien

Die Schedulingstrategien *RoundRobin* und *InactiveAging* benötigen Verwaltungsdaten, die korrekt angelegt und initialisiert werden müssen.

Verwaltungsdaten Zur Implementierung der Strategien *RoundRobin* und *InactiveAging* müssen das Alter eines Prozesses und der Wert der aktuellen Zeitscheibe gespeichert werden. Legen Sie hierzu in der Datei `os_scheduling_strategies.h` eine Struktur `struct SchedulingInformation` an, welche den Wert `timeSlice` mit geeignetem Wertebereich sowie ein Array `age` mit Werten vom Typ `Age` enthalten soll. Das Array muss Platz für genau `MAX_NUMBER_OF_PROCESSES` Elemente bieten. Benennen Sie die Struktur mittels `typedef` als `SchedulingInformation` und legen Sie eine globale Variable `schedulingInfo` vom Typ `SchedulingInformation` in `os_scheduling_strategies.c` an.

Initialisierung Implementieren Sie die folgenden Funktionen in der Datei `os_scheduling_strategies.c`, um `schedulingInfo` für *RoundRobin* und *InactiveAging* korrekt zu initialisieren. Beachten Sie bitte zum besseren Verständnis der Funktionen auch die Kommentare im Codegerüst.

`void os_resetProcessSchedulingInformation(ProcessID id):`
Diese Funktion soll das Alter des Prozesses mit Prozess-ID `id` auf `0` setzen. Fügen Sie an geeigneter Stelle in `os_exec` einen Aufruf dieser Funktion ein. Dadurch soll sichergestellt werden, dass ein neuer Prozess, dem der Index eines vorigen Prozesses zugewiesen wurde, nicht auch dessen Alter übernimmt.

`void os_resetSchedulingInformation(SchedulingStrategy strategy):`
Diese Funktion soll die zu der Strategie `strategy` gespeicherten Daten zurücksetzen. Für *RoundRobin* bedeutet dies, die Zeitscheibe auf die Priorität des derzeit ausgewählten Prozesses zu setzen. Im Fall von *InactiveAging* soll die Routine das Alter aller Prozesse auf `0` setzen. Rufen Sie diese Funktion in `os_setSchedulingStrategy` mit der neu gesetzten Strategie als Parameter auf.

2.6 Pinbelegungen

Port	Pin	Belegung
Port A	A0	LCD Pin 1 (D4)
	A1	LCD Pin 2 (D5)
	A2	LCD Pin 3 (D6)
	A3	LCD Pin 4 (D7)
	A4	LCD Pin 5 (RS)
	A5	LCD Pin 6 (EN)
	A6	LCD Pin 7 (RW)
	A7	frei
Port B	B0	frei
	B1	frei
	B2	frei
	B3	frei
	B4	frei
	B5	frei
	B6	frei
	B7	frei
Port C	C0	Button 1: Enter
	C1	Button 2: Down
	C2	Reserviert für JTAG
	C3	Reserviert für JTAG
	C4	Reserviert für JTAG
	C5	Reserviert für JTAG
	C6	Button 3: Up
	C7	Button 4: ESC
Port D	D0	frei
	D1	frei
	D2	frei
	D3	frei
	D4	frei
	D5	frei
	D6	frei
	D7	frei

Pinbelegung für Versuch 2 (*Scheduler / Stack*).