Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

Praktikum Systemprogrammierung

Versuch 5

Gemeinsamer Speicher

Lehrstuhl Informatik 11 - RWTH Aachen

6. April 2023

Commit: b3175525

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

5 Gemeinsamer Speicher

Die Interprozesskommunikation ist ein wichtiger Bestandteil von Betriebssystemen. Diese ist zum Beispiel unerlässlich für die Konstruktion komplexer Anwendungen, in denen Funktionalitäten und Aufgaben auf nebenläufige Prozesse aufgeteilt sind. In diesem Versuch werden Sie das Betriebssystem SPOS um diese Funktionalität erweitern.

5.1 Versuchsinhalte

Um Prozessen die Möglichkeit zu geben, untereinander Daten auszutauschen, wird eine Kommunikationsschnittstelle benötigt. Diese soll in SPOS mit Hilfe sogenannter *gemeinsamer Speicherbereiche* realisiert werden. Hierbei handelt es sich um allozierte Speicherbereiche, die im Gegensatz zu privatem Speicher von jedem Prozess gelesen oder beschrieben werden dürfen. In diesem Zusammenhang besteht die Hauptaufgabe des Betriebssystems darin die Konsistenz dieser Speicherbereiche während konkurrierender Zugriffe sicherzustellen. Die dazu notwendigen Mechanismen werden in den folgenden Abschnitten vorgestellt und von Ihnen implementiert.

5.2 Lernziel

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Konzepte des gemeinsamen Speichers
- Erkennung und Auflösung von Zugriffskonflikten
- Blockierte Prozesse und deren Behandlung

5.3 Grundlagen

Die in Versuch 3 (*Heap / Schedulingstrategien*) eingeführte Speicherverwaltung soll in diesem Versuch erweitert werden. Prozessen soll die Möglichkeit gegeben werden, gemeinsamen Speicher zu allozieren und darüber Daten auszutauschen. Die hierbei möglichen Zugriffskonflikte werden im folgenden Abschnitt vorgestellt. Im Anschluss daran wird erläutert, wie eine ressourcenschonende Behandlung dieser Konflikte realisiert werden kann. Außerdem wird eine weitere Schedulingstrategie eingeführt.

5.3.1 Zugriffskonflikte bei gemeinsamem Speicher

Prozesse greifen konkurrierend auf gemeinsame Speicherbereiche zu. Hierbei können folgende Zugriffskonflikte auftreten:

read-before-write: *Liest* ein Prozess i aus einem gemeinsamen Speicherbereich und wird dabei unterbrochen, so würde eine *Schreiboperation* von Prozess j auf denselben Speicherbereich das dortige Datum so verändern, dass Prozess i nach seiner Fortsetzung ein undefiniertes Datum aus dem Speicherbereich liest. Der Lesevorgang muss also vor Beginn des Schreibvorgangs abgeschlossen werden.

write-before-read: *Schreibt* ein Prozess i in einen gemeinsamen Speicherbereich und wird unterbrochen, so erhält ein Prozess j, der aus diesem Speicherbereich *liest*, ein undefiniertes Datum. Der Schreibvorgang muss also vor Beginn des Lesevorgangs abgeschlossen werden.

write-before-write: *Schreibt* ein Prozess i in einen gemeinsamen Speicherbereich und wird von einem Prozess j unterbrochen, der ebenfalls in diesen Speicherbereich *schreibt*, so ist der Inhalt des Speicherbereichs während der beiden Schreiboperationen und gegebenenfalls auch danach undefiniert. Der zweite Schreibvorgang darf also erst starten, nachdem der erste abgeschlossen ist.

Das Betriebssystem muss sicherstellen, dass ein gemeinsamer Speicherbereich, der von einem Prozess beschrieben wird, von keinem anderen Prozess beschrieben oder ausgelesen werden darf, bevor der Schreibvorgang abgeschlossen ist. Im Gegensatz dazu muss das gleichzeitige Lesen eines Speicherbereichs durch mehrere Prozesse möglich sein. Um dies sicherzustellen, muss der momentane Zustand für jeden gemeinsamen Speicherbereich bekannt sein. Erfolgt ein Lese- oder Schreibvorgang auf einen gemeinsamen Speicherbereich, so bezeichnen wir ihn als *geöffnet*. Dementsprechend wird ein Bereich, auf den momentan kein Zugriff erfolgt, als *geschlossen* bezeichnet.

5.3.2 Integration in SPOS

Um die zuvor genannten Lese- und Schreibzugriffe festzuhalten, muss das Protokoll der Allokationstabelle entsprechend erweitert werden. Für die Allokation und Freigabe eines gemeinsamen Speicherbereichs müssen die Funktionen `os_sh_malloc` bzw. `os_sh_free` bereitgestellt werden. Um den Zugriff auf gemeinsamen Speicher unter Vermeidung der zuvor beschriebenen Zugriffskonflikte zu ermöglichen, werden Prozesse die Funktionen `os_sh_read` und `os_sh_write` verwenden. Eine Leseoperation eines Prozesses erfolgt, indem der Inhalt des gemeinsamen Speicherbereichs in einen für den Prozess *privaten* Speicherbereich kopiert wird. Analog erfolgt das Beschreiben eines gemeinsamen Speicherbereichs durch das Kopieren von Daten aus einem für den Prozess privaten Speicherbereich in den gemeinsamen Speicherbereich. In Darstellung 5.1 ist ein Zugriffsbeispiel für gemeinsamen Speicher zu finden.

5.3.3 Blockierte Prozesse

Zur Lösung der Schreib- und Lesekonflikte werden Synchronisationsmechanismen benötigt, die bestimmte Zugriffe auf einen Speicherbereich zeitweise nur durch einen Prozess zulassen.
In einer Konkurrenzsituation wäre eine erste Lösung, einen Prozess aktiv warten zu lassen, bis die Zugriffsoperation des anderen beendet ist. Diese Methode verschwendet jedoch unnötig Rechenzeit und ist daher ungeeignet.
Eine zweite Lösung ist der Einsatz kritischer Bereiche. Da mit dieser Lösung das gleichzeitige Lesen eines gemeinsamen Speicherbereichs nicht möglich ist und der Scheduler für diese Zeit außer Funktion gesetzt wird, ist diese ebenfalls ungeeignet.
Eine bessere Lösung des zuvor erläuterten Problems besteht darin, dass ein blockierter Prozess bewusst auf den Rest der ihm noch zustehenden Rechenzeit verzichtet. Das Betriebssystem entzieht diesem Prozess daraufhin *sofort* die Kontrolle und verteilt die Rechenkapazität neu. Dies kann dadurch realisiert werden, dass der Scheduler manuell und nicht erst durch einen nächsten Timerinterrupt aufgerufen wird.

Lernerfolgsfragen

- Wann ist es für einen Prozess sinnvoll, seine restliche Rechenzeit abzugeben?
- Wie könnte ein konkretes Beispiel für einen blockierten Prozess aussehen?
- Auf welche Weise können Sie die vorzeitige Abgabe von Rechenzeit realisieren?

5.3.4 Referenzierung von gemeinsamem Speicher

Im folgenden Abschnitt wird erläutert, warum die Zugriffsfunktionen für den gemeinsamen Speicher über Parameter vom Datentyp `MemAddr*` an Stelle von `MemAddr` verfügen. Variablen vom Typ `MemAddr` enthalten die Adresse eines Speicherbereichs. Variablen vom Typ `MemAddr*` stellen hingegen einen Zeiger auf eine Variable vom Typ `MemAddr` dar und enthalten somit eine Adresse, an der eine Variable vom Typ `MemAddr` gespeichert ist. Die Verwendung solcher Variablen ist notwendig, da ansonsten undefinierte Zustände beim gleichzeitigen Zugriff auf gemeinsame Speicherbereiche entstehen können.
Das in Abbildung 5.1 dargestellte Beispiel verdeutlich diese Problematik. Die Prozesse 4 und 6 verfügen über einen gemeinsamen Speicherbereich. Die Startadresse dieses Bereichs ist in der Variable `shMem` gespeichert, welche wiederum an der Speicherstelle `5510` abgelegt ist. Beide Prozesse versuchen mit Hilfe der Zugriffsfunktion `os_sh_write` den gemeinsamen Speicher zu beschreiben. Der Scheduler wählt zu Beginn Prozess 4 aus. Dieser öffnet den Speicherbereich und wird unmittelbar danach durch den Scheduler unterbrochen, welcher als nächstes Prozess 6 auswählt. Dieser Prozess versucht ebenfalls

den Speicherbereich zu öffnen. Da der Speicherbereich jedoch bereits geöffnet wurde, gibt Prozess 6 seine Rechenzeit vorzeitig ab. Daraufhin erhält Prozess 4 erneut Rechenzeit, kann den Schreibvorgang durchführen und den Speicherbereich wieder schließen. Da Prozess 4 anschließend feststellt, dass der Speicherbereich nur unzureichend groß ist, gibt dieser den Speicherbereich frei, alloziert einen neuen an Adresse `1350` und aktualisiert den Inhalt der Variable `shMem` entsprechend. Wie in der Abbildung verdeutlicht, ändert sich die Adresse der Variable `shMem` im Gegensatz zu ihrem Inhalt nicht. Nun erhält Prozess 6 die Kontrolle und kann seinen Schreibvorgang korrekt durchführen.
Da der Parameter der Zugriffsfunktion `os_sh_write` vom Typ `MemAddr*` ist, kann durch Dereferenzieren dieser Variable stets die aktuelle Adresse des gemeinsamen Speicherbereichs ermittelt werden. Aus diesem Grund kann der Speicherbereich erfolgreich geöffnet, beschrieben und anschließend wieder geschlossen werden. Wäre der Datentyp des Parameters `MemAddr`, so könnte die aktuelle Adresse des gemeinsamen Speicherbereichs in der soeben beschriebenen Situation nicht mehr ermittelt werden, was einen undefinierten Zustand des Systems zur Folge hätte.

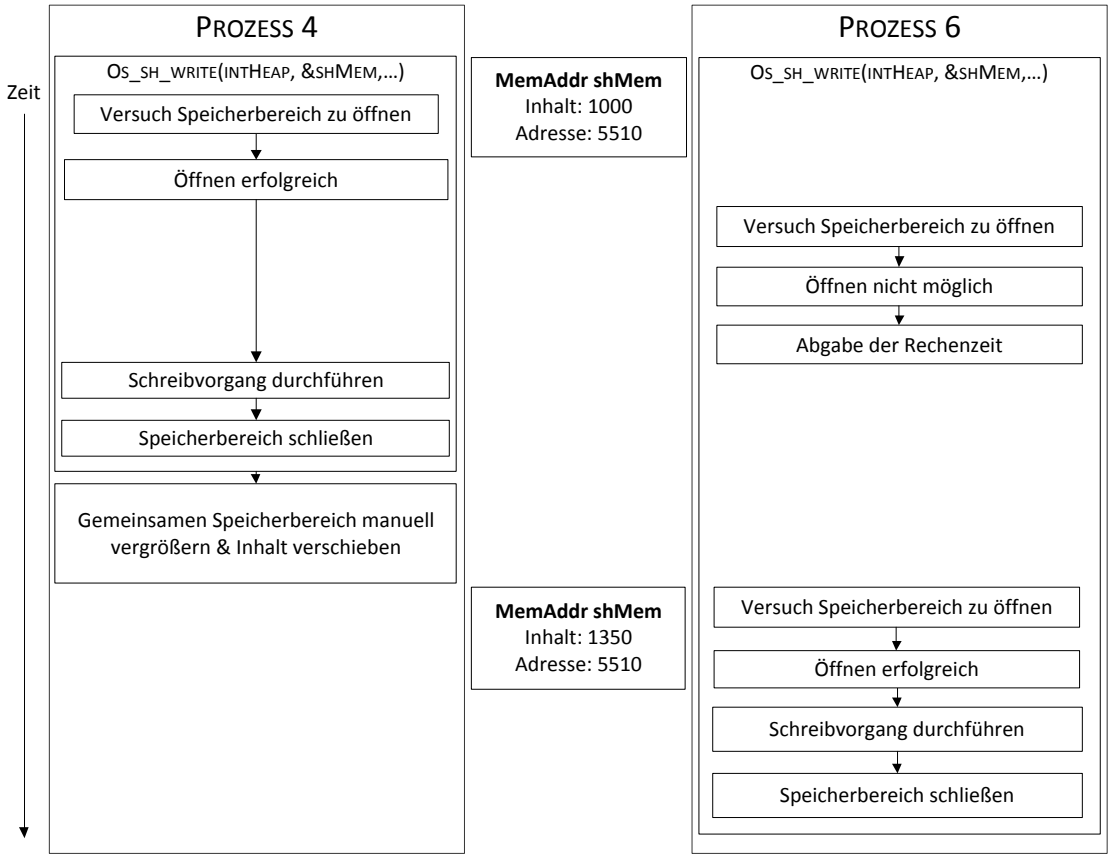

Abbildung 5.1: Prozesse 4 und 6 greifen konkurrierend auf gemeinsamen Speicher zu

Lernerfolgsfragen

- Was geschieht, wenn ein Prozess terminiert, welcher noch einen gemeinsamen Speicherbereich geöffnet hat?

5.3.5 Multilevel-Feedback-Queue

Die *Multilevel-Feedback-Queue* ist eine dynamische Schedulingstrategie, die auch in gängigen Betriebssystemen Verwendung findet. Sie verwaltet Prozesse in Prioritätsklassen. Innerhalb einer Klasse werden die Prozesse in eine Warteschlange eingeordnet und ähnlich zur Round-Robin Strategie ausgewählt. Die nutzbaren Zeitscheiben werden hierbei allerdings für jeden Prozess einzeln verwaltet und sind initial klassenabhängig. Dabei gilt: je höher die Klassen-Priorität, desto weniger Zeitscheiben stehen einem Prozess zur Verfügung. Ausgewählt wird der erste verfügbare Prozess der am höchsten priorisierten, nicht-leeren Warteschlange. Im Folgenden beziehen sich die Begriffe hoch und niedrig immer auf die jeweilige Klassen-Priorität.

Das Einordnen eines Prozesses sieht dabei wie folgt aus:
Zu Beginn wird ein Prozess entsprechend seiner Priorität in eine Klasse eingefügt und in die zugehörige Warteschlange eingereiht. Dabei erhält dieser initial die für diese Klasse definierte Anzahl an Zeitscheiben. Wird der Prozess nun im Laufe des Schedules ausgewählt, verringert sich die Anzahl der restlichen Zeitscheiben um eins. Mit jedem Aufruf des Schedulers wird erneut geprüft, ob eventuell ein Prozess in einer höher priorisierten Klasse bereit zur Ausführung ist. Hat ein Prozess seine Zeitscheiben aufgebraucht, wird er in die Warteschlange der nächstniedrigeren Klasse eingefügt und die Anzahl der Zeitscheiben entsprechend der neuen Klasse initialisiert. Befindet sich der Prozess bereits in der niedrigsten Klasse, verbleibt er in dieser und wird wieder hinten eingereiht. Gibt ein Prozess frühzeitig Rechenzeit ab, wird dieser mit seinen verbleibenden Zeitscheiben ans Ende derselben Warteschlange angehängt. Falls ein Prozess frühzeitig Rechenzeit abgibt, aber keine verbleibenden Zeitscheiben mehr hat, wird dieser wie gewohnt in die niedrigere Klasse eingefügt.

Außerdem gilt weiterhin, dass der Leerlaufprozess nur dann durch eine Schedulingstrategie ausgewählt werden soll, falls kein anderer Prozess zur Ausführung bereit ist. Dies ist insbesondere dann relevant, falls ein Prozess gestartet wird, nachdem kein Prozess aktiv war.

Die Strategie *Multilevel-Feedback-Queue* wird in Abbildung 5.2 am Beispiel zweier Prozesse erläutert. In diesem Beispiel verwaltet die Strategie lediglich drei Prioritätsklassen mit 2, 4 und 8 Zeitscheiben. Dabei ist zu Beginn nur Prozess 1 gestartet, der durch

Schritt	Q1(2)	Q2(4)	Q3(8)	Running	Ereignis
0					1 startet
1		1(4)		1	2 startet
2	2(2)	1(3)		2	
3	2(1)	1(3)		2	
4		1(3),2(4)		1	1 yield
5		2(4),1(2)		2	
6		2(3),1(2)		2	
...	...	...	...	...	

Abbildung 5.2: Veranschaulichung der Schedulingstrategie Multilevel-Feedback-Queue

seine Priorität der Klasse 2 zugeordnet wird und dem somit initial vier Zeitscheiben zur Verfügung stehen. Dieser Prozess startet in seiner ersten Zeitscheibe Prozess 2 mit höherer Priorität und gibt in seiner zweiten Zeitscheibe vorzeitig Rechenzeit ab. Anhand der Klammern lässt sich die Anzahl der verbleibenden Zeitscheiben eines Prozesses ablesen.

5.4 Hausaufgaben

Achtung

Passen Sie das `define VERSUCH` auf die aktuelle Versuchsnummer an.

Implementieren Sie die in den nächsten Abschnitten beschriebenen Funktionalitäten. Halten Sie sich an die hier verwendeten Namen und Bezeichnungen für Variablen, Funktionen und Definitionen.
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 sowie die Testtasks mit aktivierten Compileroptimierungen bestehen. Wie Sie die Optimierungen einschalten ist dem begleitenden Dokument in Abschnitt *6.3.2 Probleme bei der Speicherüberwachung* zu entnehmen.

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.

Achtung

Achten Sie darauf, wann Sie einen Zeiger und wann Sie eine Variable als Parameter nutzen müssen. Die Übersicht in Kapitel 5.4.4 soll Ihnen dazu als Hilfe dienen. Als weitere Hilfe können Sie die Doxygen-Dokumentation nutzen.

Hinweis

Das Ändern des `defines VERSUCH` wird zu Beginn eine Fehlermeldung hervorrufen. Dies liegt daran, dass der Taskmanager ab Versuch 5 die Schedulingstrategie Multi-Level-Feedback-Queue verwendet. Um diesen Fehler zu beheben, muss das enum `SchedulingStrategy` um den Eintrag `OS_SS_MULTI_LEVEL_FEEDBACK_QUEUE` erweitert werden.

5.4.1 Vorzeitige Abgabe der Rechenzeit eines Prozesses

Implementieren Sie die Funktion `os_yield`, mit der ein Prozess seine übrige Rechenzeit abgeben kann. Zu diesem Zweck soll die Funktion den Scheduler unmittelbar aufrufen. Nutzen Sie hierfür den Ausdruck `TIMER2_COMPA_vect()`, welcher einen Funktionsaufruf der Interruptserviceroutine des Schedulers darstellt.
Ein Prozess, welcher die Funktion `os_yield` aufruft, soll nicht im darauffolgenden Scheduleraufruf ausgewählt werden. Weisen Sie zu diesem Zweck dem Prozess, der die Rechenzeit abgibt, den Status `OS_PS_BLOCKED` zu. Der Scheduler und die Schedulingstrategien müssen von Ihnen so angepasst werden, dass Prozesse mit diesem Status *einmalig* ignoriert werden.
Implementieren Sie `os_yield` als kritischen Bereich. Stellen Sie vor dem manuellen Aufruf des Schedulers außerdem sicher, dass der Scheduler auch weiterhin automatisch durch Timerinterrupts aufgerufen wird. Speichern Sie außerdem lokal den Zustand der kriti-

schen Bereiche. Hierzu muss zum einen die Anzahl an geöffneten kritischen Sektionen gesichert werden und zum anderen der Status des Global Interrupt Enable Bits (`GIEB`) des `SREG`-Registers.
Sobald wieder in `os_yield` zurückgekehrt wird, muss der zuvor gespeicherte Zustand wiederhergestellt werden. Es ist sinnvoll, die Funktion `os_yield` nicht nur bei Schreib- und Leseblockaden, sondern auch an anderen Stellen im Betriebssystem zu verwenden. Die Funktion kann zum Beispiel die Warteschleife des Dispatchers bzw. die in der Funktion `os_kill` ersetzen. Setzen Sie dies um, aber beachten Sie, dass `os_yield` für den Fall der Terminierung angepasst werden muss.
Für eine effizientere Implementierung soll in dem Fall, das kein Prozess den Status `OS_PS_READY` und ein Prozess den Status `OS_PS_BLOCKED` hat, der geblockte Prozess erneut ausgewählt werden.

Achtung

Alle Schedulingstrategien sollten erneut auf Funktionalität überprüft werden. Es soll zu **KEINEM** Zeitpunkt der Leerlaufprozess ausgewählt werden, solange mindestens ein Prozess mit Status `OS_PS_READY` **ODER** `OS_PS_BLOCKED` existiert.

5.4.2 Multilevel-Feedback-Queue

Erstellen einer Warteschlange

Um die `Multilevel-Feedback-Queue` realisieren zu können, müssen zunächst weitere Verwaltungsstrukturen angelegt werden. Die Warteschlangen der einzelnen Klassen sollen durch einen Ringpuffer realisiert werden. Legen Sie dazu in der Datei `os_scheduling_strategies.h` ein `struct ProcessQueue` an, das ein ausreichend großes Array zum Speichern von Prozess-IDs `data` und die maximale Anzahl an IDs `size` speichert. Zusätzlich soll eine `ProcessQueue` zwei Felder `head` und `tail` enthalten, die als Indizes in diesem Speicherbereich dienen. Dabei sollen neue ProzessIDs an Index `head` angehängt und dieser inkrementiert werden. Umgekehrt werden Elemente an der Position `tail` entnommen und diese dann inkrementiert werden. Somit läuft der Index `tail` dem Index `head` hinterher. Alle Prozess-IDs, die sich zwischen `head` und `tail` befinden, sind somit noch in der Warteschlange eingereiht. Um den Puffer leichter benutzen zu können, implementieren Sie die folgenden Zugriffsfunktionen in der Datei `os_scheduling_strategies.c`. Diese erhalten jeweils einen Zeiger auf eine `ProzessQueue`, da die Funktionalität für mehrere Instanzen des Puffers genutzt werden soll.
Die im Folgenden genannten Funktionen, welche die `Multilevel-Feedback-Queue`-Implementierung betreffen, sollen öffentlich gemacht werden, um eine Verwendung in den Testtasks zu gewährleisten.

Puffer anlegen Initialisieren Sie in der Funktion `pqueue_init` den Eingabepuffer. Speichern Sie anschließend die Größe des Puffers in `size` und initialisieren Sie die Felder `head` und `tail` mit 0.

Puffer zurücksetzen Setzen Sie in der Funktion `pqueue_reset` den Eingabepuffer zurück, indem Sie `head` und `tail` auf 0 setzen.

Sicherheitsabfrage Überprüfen Sie in der Funktion `pqueue_hasNext`, ob sich noch Prozess-IDs im Speicher befinden.

Lesen und Schreiben Implementieren Sie zum Lesen und Schreiben auf den Ringpuffer die Funktionen `pqueue_getFirst`, `pqueue_dropFirst` und `pqueue_append`. Dabei sollte `pqueue_getFirst` nur das erste Element zurückliefern und nicht aus der Warteschlange entfernen.

Suchen und Entfernen Implementieren sie eine Funktion `pqueue_removePID`, welche die übergebene Prozess-ID aus der Queue entfernt, falls diese existiert. Die Reihenfolge der anderen Queue-Elemente soll dabei erhalten bleiben.

Erweitern der Schedulinginformationen

Fügen Sie ein Array für die Verwaltung der individuellen Zeitscheiben der Prozesse der Struktur `SchedulingInfo` hinzu. Des Weiteren werden dort auch die Warteschlangen gespeichert. Unsere Implementierung soll die Prozesse in vier verschiedenen Klassen mit jeweils 1, 2, 4 und 8 Zeitscheiben verwalten. Die initiale Zuordnung der Prozesse zu einer Warteschlange erfolgt dabei über die beiden MSBs der Priorität. Dabei entspricht `11` der höchsten Klasse und `00` der niedrigsten.
Erstellen Sie in der Datei `os_scheduling_strategies.c` eine Funktion `MLFQ_getQueue`, die einen Index erhält und einen Zeiger auf die entsprechende Warteschlange zurückliefert. Somit wird der Zugriff auf eine bestimme Warteschlange leichter. Implementieren Sie außerdem die Funktion `os_initSchedulingInformation`, die die Warteschlangen initialisiert. Rufen Sie letztere an geeigneter Stelle auf.
Beachten Sie, dass die Funktionen `os_resetProcessSchedulingInformation` und `os_resetSchedulingInformation` angepasst werden müssen, damit unter anderem das Starten eines neuen Prozesses und der Wechsel zwischen Schedulingstrategien einwandfrei funktionieren.

Implementierung der Strategie

Implementieren Sie nun die Schedulingstrategie wie im Grundlagenkapitel beschrieben. Die Funktion soll `os_Scheduler_MLFQ` heißen und analog zu den anderen Strategien aufgebaut sein. Integrieren Sie Ihre Implementierung anschließend in den Scheduler und fügen Sie dem enum `SchedulingStrategy` den Wert `OS_SS_MULTI_LEVEL_FEEDBACK_QUEUE` hinzu.

5.4.3 Gemeinsamer Speicher

Der Datenaustausch zwischen Prozessen soll über gemeinsam genutzte Speicherbereiche erfolgen. Erweitern Sie dazu die Datei `os_memory.c` wie nachfolgend beschrieben.
Überlegen Sie sich zunächst eine geeignete Erweiterung des in der Allokationstabelle verwendeten Protokolls. Die neu benötigten Zustände sollen neben den Zuständen aus Versuch 3 (*Heap / Schedulingstrategien*) weiterhin mit insgesamt vier Bit dargestellt werden. Hierbei müssen gemeinsame Speicherbereiche in der Allokationstabelle nicht nur als solche markiert, sondern auch bestehende Lese- und Schreibzugriffe festgehalten werden. Ihre Implementierung soll das gleichzeitige Lesen von mindestens zwei Prozessen erlauben. Bedenken Sie, dass für gemeinsame Speicherblöcke nicht die Prozess-ID des Prozesses gespeichert werden muss, welcher den Speicherbereich alloziert hat. Beachten Sie außerdem, dass ein Speicherbereich nach Beendigung eines Lesevorgangs aufgrund anderer Lesezugriffe weiterhin gesperrt sein könnte. Dies muss ebenfalls durch Ihr Protokoll abgedeckt sein.
Berücksichtigen Sie die nachfolgenden Anforderungen und Schnittstellenbeschreibungen. Eine exakte Auflistung der Beschreibung der Schnittstellen finden Sie im Kapitel 5.4.4.

> **Achtung**
>
> Einige der Funktionen, die in diesem Abschnitt vorgestellt werden, stellen kritische Bereiche dar, die nicht durch andere Prozesse unterbrochen werden dürfen. Verwenden Sie also an sinnvollen Stellen die Systemfunktionen zur Verwaltung kritischer Bereiche.

Implementieren Sie die Funktionen `os_sh_malloc` und `os_sh_free`. Orientieren Sie sich hierbei an den entsprechenden Implementierungen der Funktionen `os_malloc` und `os_free` aus Versuch 3. Die Funktion `os_sh_malloc` soll mit dem entsprechenden Heaptreiber und der Anzahl zu allozierender Bytes als Parameter aufgerufen werden und eine Variable vom Typ `MemAddr` zurückgeben, welche die Adresse des Beginns des allozierten Speicherbereichs enthält. Der Funktion `os_sh_free` soll ein Heaptreiber und ein Zeiger auf einen `MemAddr`-Datentyp als Parameter übergeben werden und den dazugehörigen Speicherbereich wieder freigeben. Für den Fall, dass der freizugebende Speicherbereich noch geöffnet ist, muss die Funktion so lange warten, bis der entsprechende Speicherbereich geschlossen wurde. Setzen Sie hierfür die zuvor implementierte Funktion `os_yield` ein. Beachten Sie in diesem Zusammenhang die im Vergleich zur Funktion `os_free` abweichende Funktionssignatur. Sollte der zugehörige Speicherblock kein gemeinsamer Speicher sein, so soll die Funktion darauf mittels einer Fehlermeldung reagieren und den Speicher unverändert lassen. Beachten Sie dabei, dass analog dazu `os_free` keinen gemeinsamen Speicherbereich freigeben darf. In einem solchen Fall soll ebenfalls eine Fehlermeldung ausgegeben werden. Die Funktion `os_realloc` muss von Ihnen nicht angepasst werden.

Speicherblockierung

Für den Zugriff auf den gemeinsamen Speicher sollen Prozesse die Funktionen `os_sh_read` und `os_sh_write` aufrufen, welche das korrekte Lesen und Beschreiben von gemeinsamen Speicherbereichen unter konkurrierenden Zugriffen sicherstellen.
Da diese Funktionen die genannten Zugriffskonflikte berücksichtigen müssen, werden zunächst die Hilfsfunktionen `os_sh_readOpen`, `os_sh_writeOpen` und `os_sh_close` eingeführt, welche die Lese- und Schreibsperren realisieren. Ein Aufruf von `os_sh_readOpen` oder `os_sh_writeOpen` öffnet den Speicherbereich zum Lesen bzw. Schreiben. Ein Aufruf von `os_sh_close` schließt diesen wieder. Stellen Sie sicher, dass die im Grundlagenkapitel erläuterten Zugriffskonflikte korrekt behandelt werden. Diese Funktionen verhindern beispielsweise, dass ein Prozess in einen Speicherbereich schreibt, während ein anderer Prozess auf diesen zugreift. Das parallele Auslesen von Speicherbereichen durch mindestens zwei Prozessen muss jedoch unterstützt werden. Für den Fall, dass ein Speicherbereich nicht geöffnet werden kann, soll eine frühzeitige Rechenzeitabgabe erfolgen. Nachdem die Funktionen `os_sh_readOpen` und `os_sh_writeOpen` einen Speicherbereich geöffnet haben, liefern diese das dereferenzierte Argument `ptr` zurück. Achten Sie bei der Wahl des Zeitpunkts für die Dereferenzierung darauf, dass die Adresse der tatsächlich geöffneten Speicherstelle entspricht und die Dereferenzierung nicht außerhalb eines kritischen Bereiches stattfindet.

> **Hinweis**
>
> Die genannten Hilfsfunktionen sollten im Allgemeinen vor einem Aufruf durch Anwendungsprogramme geschützt werden. Um die Überprüfung Ihrer Methoden während des Versuches zu ermöglichen, müssen diese jedoch für jeden Prozess sichtbar deklariert werden.

Lese- und Schreibzugriffe

Der Zugriff auf den gemeinsamen Speicher wird durch die Funktionen `os_sh_read` und `os_sh_write` realisiert. Diese sollen die Funktionen `os_sh_readOpen`, `os_sh_writeOpen` und `os_sh_close` geeignet aufrufen, um sicherzustellen, dass die aus den genannten Zugriffskonflikten resultierenden Lese- und Schreibabhängigkeiten korrekt behandelt werden. Bedenken Sie, dass der eigentliche Lese- oder Schreibvorgang durch den Scheduler unterbrechbar sein soll. Verwenden Sie kritische Bereiche lediglich an notwendigen Stellen innerhalb der Open-Funktionen.

`void os_sh_read(Heap const* heap, MemAddr const* ptr, uint16_t offset, MemValue* dataDest, uint16_t length):`
Diese Funktion soll die ersten `length` Bytes, von *Beginn* des gemeinsamen Speicherbereichs an, nach `dataDest` kopieren. Die Adresse des gemeinsamen Speicherbereichs kann

mit Hilfe des Parameters `ptr` ermittelt werden und befindet sich auf dem Heap, der durch den Parameter `heap` referenziert wird. Die Variable `dataDest` bezieht sich immer auf eine Adresse im internen Speicher. Soll nicht von Beginn des gemeinsamen Speicherbereichs aus kopiert werden, kann eine Verschiebung mittels der Variable `offset` angegeben werden. In diesem Fall werden `length` Bytes beginnend bei der Startadresse des gemeinsamen Speicherbereichs plus `offset` nach `dataDest` kopiert.

`void os_sh_write(Heap const* heap, MemAddr const* ptr, uint16_t offset, MemValue const* dataSrc, uint16_t length):`
Diese Funktion soll die ersten `length` Bytes von der Adresse `dataSrc` in den gemeinsamen Speicherbereich kopieren. Die Variable `dataSrc` bezieht sich auf eine Adresse im internen Speicher. Analog zur Funktion `os_sh_read` wird der gemeinsame Speicherbereich beginnend bei seiner Startadresse beschrieben. Soll der gemeinsame Speicher nicht von Beginn an beschrieben werden, so kann mit Hilfe des Parameters `offset` eine Verschiebung angegeben werden. In diesem Fall werden `length` Bytes aus `dataSrc` aus kopiert und diese beginnend bei der Startadresse des gemeinsamen Speicherbereichs plus `offset` abgespeichert.
Bei der Benutzung von `os_sh_read` und `os_sh_write` muss überprüft werden, ob ein Zugriff außerhalb der Grenzen eines gemeinsamen Speicherbereichs stattfindet. In einem solchen Fall ist ein Fehler auszugeben und der Speicherbereich unverändert zu belassen.

In Listing 5.1 finden Sie ein exemplarisches Beispiel für den Zugriff auf gemeinsamen Speicher. Der Prozess legt zunächst den durch `ptr` referenzierten gemeinsamen Speicherbereich mit der Länge drei Bytes (1) und den lokalen Speicherbereich `num` mit der Länge zwei Bytes an (2). Die zwei Bytes von `num` werden mit den `uint8_t`-Werten `34` und `17` initialisiert (2). Anschließend wird eine lokale Variable `num2` vom Typ `uint8_t` mit dem Wert `8` angelegt (3). Zusätzlich wird `res` zur Zwischenspeicherung des Ausgabewertes deklariert (3). Zuletzt wird der private Speicherbereich `control` mit einer Größe von drei Bytes angelegt (4). Nach dem Anlegen der Variablen werden zwei Schreibzugriffe (5, 6) und ein Lesezugriff (7) auf dem gemeinsamen Speicher durchgeführt. Zwei Werte aus `control` werden in `res` aufsummiert (8, 9) und anschließend ausgegeben (10).

```
1  MemAddr ptr = os_sh_malloc(intHeap, 3);
2  uint8_t num[] = {34, 17};
3  uint8_t num2 = 8; uint8_t res = 0;
4  MemAddr control = os_malloc (intHeap, 3);
5  os_sh_write(intHeap, &ptr, 0, num, 2);
6  os_sh_write(intHeap, &ptr, 2, &num2, 1);
7  os_sh_read(intHeap, &ptr, 0, (MemValue*)control, 3);
8  res = intHeap->driver->read(control);
9  res += intHeap->driver->read(control+2);
10 lcd_writeDec(res);
```

Listing 5.1: Zugriffsbeispiel auf den gemeinsamen dynamischen Speicher

Lernerfolgsfragen

- Was wird im Zugriffsbeispiel in Listing 5.1 in den gemeinsamen Speicher geschrieben?
- Welche Zeichenfolge wird in der letzten Zeile des Zugriffsbeispiels in Listing 5.1 auf dem Display ausgegeben?
- Welche Randbedingungen müssen die Funktionen `os_sh_read` und `os_sh_write` jeweils beachten?

5.4.4 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_memory:`
  - ☐ Allokation und Freigabe:
    - ☐ `MemAddr os_sh_malloc(Heap* heap, uint16_t size)`
    - ☐ `void os_sh_free(Heap* heap, MemAddr* ptr)`
  - ☐ Öffnen und Schließen:
    - ☐ `MemAddr os_sh_readOpen(Heap const* heap, MemAddr const *ptr)`
    - ☐ `MemAddr os_sh_writeOpen(Heap const* heap, MemAddr const *ptr)`
    - ☐ `void os_sh_close(Heap const* heap, MemAddr addr)`
  - ☐ Lesen und Schreiben:
    - ☐
      ```
      void os_sh_write(Heap const* heap,
                       MemAddr const* ptr,
                       uint16_t offset,
                       MemValue const* dataSrc,
                       uint16_t length)
      ```
    - ☐
      ```
      void os_sh_read(Heap const* heap,
                      MemAddr const* ptr,
                      uint16_t offset,
                      MemValue* dataDest,
                      uint16_t length)
      ```
- ☐ `os_scheduler:`
  - ☐ Vorzeitige Abgabe der Rechenzeit:
    - ☐ `void os_yield(void)`
- ☐ `os_scheduling_strategies:`
  - ☐ Vorzeitige Abgabe der Rechenzeit:
    - ☐ Anpassung der Schedulingstrategien
  - ☐ Datentypen
    - ☐ `struct ProcessQueue`

- Funktionen
  - `void pqueue_init(ProcessQueue *queue)`
  - `void pqueue_reset(ProcessQueue *queue)`
  - `bool pqueue_hasNext(const ProcessQueue *queue)`
  - `ProcessID pqueue_getFirst(const ProcessQueue *queue)`
  - `void pqueue_dropFirst(ProcessQueue *queue)`
  - `void pqueue_append(ProcessQueue *queue, ProcessID pid)`
  - `void pqueue_removePID(ProcessQueue *queue, ProcessID pid)`
  - `ProcessQueue* MLFQ_getQueue(uint8_t queueID)`
  - `void os_initSchedulingInformation`
  - `ProcessID os_Scheduler_MLFQ(Process const processes[], ProcessID current)`

5.5 Präsenzaufgaben

5.5.1 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.

5.6 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	SIO2*
	B4	\CS
	B5	SI
	B6	SO
	B7	SCK
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 5 (*Gemeinsamer Speicher*).
* Nicht für den Versuch relevant.