RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHRSTUHL
INFORMATIK 11
SOFTWARE FÜR
EINGEBETTETE
SYSTEME

Praktikum Systemprogrammierung

Versuch 3

Heap / Schedulingstrategien

Lehrstuhl für Informatik 11 - RWTH Aachen

14. April 2011

Inhaltsverzeichnis

Dieses Dokument ist Teil der begleitenden Unterlagen zum *Praktikum Systemprogrammierung*. Alle zu diesem Praktikum benötigten Unterlagen stehen im L^{2}P-Lernraum unter http://www.elearning.rwth-aachen.de zum Download bereit.

Folgende Emailadresse ist für Kritik, Anregungen oder Verbesserungsvorschläge verfügbar:

psp@embedded.rwth-aachen.de

3 Heap / Schedulingstrategien

In diesem Praktikumsversuch werden verschiedene Speicherarten des ATmega 644 erläutert und die Verwaltung des privaten Speichers für SPOS implementiert. Der hier eingeführte private dynamische Speicher stellt (parallel zum Stack) eine zusätzliche Art von Speicher zur Verfügung, der von Prozessen zur Laufzeit allokiert bzw. deallokiert werden kann. Außerdem wird der im vergangenen Versuch entwickelte Scheduler um weitere Strategien ergänzt, die die Auswahl des nächsten Prozesses anhand unterschiedlicher Algorithmen treffen.

3.1 Versuchsinhalte

Nach der Durchführung des vorangegangenen Versuchs unterstützt SPOS die (implizite) Allokation von Stackspeicher für Anwendungsprogramme. Dieser Stackspeicher wird vom Mikrocontroller selbst verwaltet und wird z. B. für lokale Variablen und Funktionsaufrufe benötigt. Die meisten Betriebssysteme bieten dem Entwickler die Möglichkeit, zusätzlichen eigenen Speicher (explizit) zu allokieren. In der Regel muss der Entwickler der Anwendungsprogramme selbst auf die Freigabe des von ihm angeforderten Speichers achten.

In den nächsten Abschnitten werden zunächst die theoretischen Grundlagen und Besonderheiten des ATmega 644 erklärt. In den Abschnitten 3.3.2 und 3.3.3 werden unterschiedliche Speicherarten des ATmega 644 vorgestellt und der Unterschied eines dynamischen Speichers zum Stack gezeigt. Kapitel 3.4 beschäftigt sich genauer mit den einzelnen Aufgabestellungen und der konkreten Implementierung der benötigten Funktionalität.

3.2 Lernziel

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Schedulingstrategien
- Verschiedene Speicherarten des ATmega 644
- Dynamischer Speicher (Heap)
- Garbage Collection

3.3 Grundlagen

In diesem Abschnitt werden weitere Schedulingstrategien für den in Versuch 2 entwickelten Scheduler eingeführt. Im Anschluss daran werden die Grundlagen der dynamischen Speicherverwaltung vorgestellt. Dazu gehört ein Überblick über drei verschiedene Speicherarten des ATmega 644, sowie die Einführung in die dynamische Allokation und Freigabe von Ressourcen.

3.3.1 Schedulingstrategien

Eine Schedulingstrategie wird zur Bestimmung des nächsten Prozesses, der die Kontrolle über das System erhält, genutzt. Schedulingstrategien können durch verschiedene Eigenschaften beschrieben werden. Diese sind z. B. Fairness, Effizienz, Durchsatz oder Vorhersagbarkeit. Im Rahmen des Praktikums liegt der Schwerpunkt in der Implementierung der Strategien, daher wird für die theoretischen Hintergründe der Schedulingstrategien auf die passende Vorlesung *Betriebssysteme und Systemsoftware* verwiesen. In diesem Versuch werden drei Strategien vorgestellt, die in SPOS benutzt werden. Die Beschreibung der Strategien *Even* und *Random* wurde ausgelassen, da diese bereits Teil des vorherigen Versuchs waren.

Achtung

Aufgrund der hier vorgeschlagenen Implementierungsweise der Schedulingstrategien muss die Priorität jedes aktiven Prozesses echt größer Null sein.

Run-To-Completion

Eine sehr populäre Schedulingstrategie bei nicht-präemptiven Systemen[1] ist die *Run-To-Completion* Strategie. Hierbei wird der aktuell laufende Prozess solange beibehalten, bis dieser terminiert. Dabei muss der Programmierer der Anwendungsprogramme selbst darauf achten, keine langlebigen Prozesse zu erzeugen, da ansonsten andere Prozesse unter Umständen nicht (rechtzeitig) ausgeführt werden. Häufig wird diese Strategie in Form einer Warteschlange implementiert. Generell ist nur notwendig, dass jeder wartende Prozess irgendwann ausgeführt wird – unter der Voraussetzung, dass alle Prozesse terminieren. Falls, wie bei SPOS, ein Leerlaufprozess (Idle-Prozess) existiert ist darauf zu achten, dass dieser nur dann aktiv ist, falls kein Anwendungsprozess auf Ausführung wartet. Dies ist inbesondere relevant, falls ein Anwendungsprozess gestartet wird, nachdem kein kein Anwendungsprozess aktiv war.

[1] Systeme die keine *vorzeitige Unterbrechung* des laufenden Prozesses erzwingen und daher häufig mit nur einem einzigen Stack auskommen.

RoundRobin

Der nächste aktive Prozess i erhält eine Zeitscheibe zugeteilt. Diese Zeitscheibe wird vom Scheduler mit seiner Priorität $Priority_i$ initialisiert und bei jedem weiteren Scheduleraufruf um eins verringert. Erreicht die Zeitscheibe des Prozesses i den Wert 0, wählt der Scheduler den nächsten wartenden Prozess j aus und teilt diesem die Zeitscheibe der Größe $Priority_j$ zu. Die Prozesse werden folglich, ähnlich der *Even* Strategie, der Reihe nach abgearbeitet und der neu ausgewählte Prozess bleibt solange aktiv, bis seine Zeitscheibe abgelaufen ist.

InactiveAging

Zusätzlich zur Priorität erhält jeder Prozess i bei dieser Strategie ein Alter, welches mit dem Wert von $Priority_i$ initialisiert wird. Die Kontrolle wird immer dem ältesten Prozess übergeben. Mit jedem Aufruf des Schedulers wächst das Alter jedes inaktiven (wartenden) Prozesses um seine Priorität, das Alter des aktiven Prozesses bleibt hingegen unverändert. Hochpriorisierte inaktive Prozesse altern daher schneller als Prozesse mit niedriger Priorität, und bekommen somit öfter Rechenzeit zugeteilt.

Diese Strategie kann wie folgt schrittweise beschrieben werden: Der Scheduler erhöht als erstes das Alter aller inaktiven Prozesse um deren Priorität, wählt dann den ältesten Prozess i aus und setzt dessen Alter auf den Wert von $Priority_i$ zurück. Kann i nicht eindeutig bestimmt werden, wird aus den ältesten Prozessen der höchstpriorisierte ausgewählt. Ist auch diese Wahl nicht eindeutig, wird daraus der Prozess mit der kleinsten PID gewählt.

Hinweis

- Das Alter des Prozesses muss in einer hinreichend dimensionierten Variable gespeichert werden, damit bei seiner Erhöhung keine ungewollten Überläufe auftreten.
- Die für die Strategie *RoundRobin* genutzte Zeitscheibe kommt bei *InactivAging* nicht zum Einsatz.

Die Strategie *InactiveAging* wird in Abbildung 3.1 am Beispiel dreier Prozesse mit den Prioritäten 2, 5 und 17 verdeutlicht. Die zu dem jeweiligen Zeitpunkt laufenden Prozesse sind mit einem Kreis versehen. 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 an die Reihe kommt. Die Werte an den Pfeilen geben das neue Alter des jeweiligen Prozesses vor dem Zurücksetzen auf die jeweilige Priorität an.

Zu Beginn ist das Alter aller drei Prozesse mit ihrer jeweiligen Priorität initialisiert. Die folgende Auflistung zeigt den Prozesswechsel in mehreren aufeinander folgenden

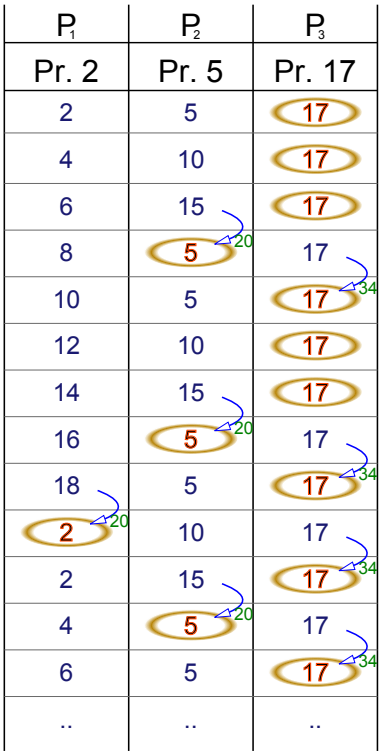

Abbildung 3.1: Veranschaulichung der Schedulingstrategie InactiveAging

Scheduleraufrufen:

1. Der älteste Prozess ist P_3 mit dem Alter 17, daher wird dieser vom Scheduler zuerst ausgewählt.

2. Die Prozesse P_1 und P_2 altern zum nächsten Scheduleraufruf jeweils um ihre Priorität (2 bzw. 5) auf das Alter 4 bzw. 10. P_3 altert nicht, weil er aktiv ist.

3. Analog altern P_1 und P_2 im dritten Scheduleraufruf auf 6 bzw. 15. P_3 behält das Alter 17 und bleibt aktiv (wegen $17 > 15$ und $17 > 6$).

4. Im nächsten Aufruf des Schedulers wächst das Alter von P_2 auf den Wert 20 und überschreitet das Alter von P_3. Deswegen wird P_2 als aktiver Prozess ausgewählt und sein Alter auf die eigene Priorität zurückgesetzt.

5. Im fünften Scheduleraufruf ändert sich nur das Alter von P_1: P_2 war aktiv, also ist keine Änderung nötig und das Alter von P_3 ändert sich zwischenzeitlich auf 34 und übersteigt damit das von P_2 (welches 5 ist) und das von P_1 (10). Daher wird P_3 als nächster aktiver Prozess gewählt und sein Alter auf 17 zurückgesetzt.

6. Und so weiter.

Lernerfolgsfragen

- Was ist eine Schedulingstrategie? Wann wird sie verwendet?
- Welche Eigenschaften einer Schedulingstrategie sind wichtig?
- Sei N die Anzahl momentan laufender Prozesse. Warum kann bei der Implementierung von *RoundRobin* oder *Even* nicht davon ausgegangen werden, dass der Prozess mit PID $[(i+1) \bmod N]$ immer ausgewählt werden kann, sobald i abgearbeitet ist oder unterbrochen wird?
- Welche Auswirkungen hätte die Abgabe der Kontrolle an einen unbenutzten Prozess-Slot in `os_processes`?
- Welche Vorteile bietet es, bei *InactiveAging* den Leerlaufprozess nicht mitaltern zu lassen?
- Welche Vor- und Nachteile sind bei *RoundRobin* im Vergleich zu *InactiveAging* erkennbar?
  - Welche dieser Strategien ist fair (mit Berücksichtigung der Prioritäten als Gewichtung)?
  - Wie ändert sich jeweils die Ausführung, wenn man alle Prioritäten mit einem festen Faktor multipliziert?

3.3.2 Speicher des ATmega 644

Aktuelle Mikrocontroller verfügen über verschiedene Arten von Speicher, die sich im internen Aufbau und typischer Verwendung unterscheiden. Der ATmega 644 bietet seinem Benutzer drei verschiedene Speicherarten, die für unterschiedliche Zwecke genutzt werden: 2 kByte EEPROM, 64 kByte Flash-Speicher und 4 kByte SRAM. Die Merkmale einzelner Speicherarten werden in den folgenden Abschnitten genauer erläutert.

EEPROM

EEPROM steht für Electrically Erasable Programmable Read Only Memory. Dies ist ein nichtflüchtiger Speicher, der seinen Zustand nach dem Ausschalten der Betriebsspannung behält. Das Beschreiben dieses Speichers ist im ATmega 644 auf etwa 100.000 Schreib- und Löschzyklen beschränkt, danach kann es zu Datenverlusten kommen. Gespeicherte Daten können aus dem EEPROM jedoch beliebig oft ausgelesen werden. Aus diesem Grund eignet sich EEPROM am besten zum Speichern von Programmeinstellungen.

Flash-ROM

Flash-ROMs sind - ähnlich zu EEPROMs - nichtflüchtige Datenspeicher, die elektrisch lösch- und beschreibbar sind. Sie können beliebig oft ausgelesen, aber nicht beliebig oft beschrieben werden. Im Fall des ATmega 644 liegt die Grenze bei etwa 10.000 Schreib- und Löschzyklen. In einem Flash-ROM können im Gegensatz zum EEPROM nicht einzelne Bytes gelöscht und neu beschrieben werden, sondern es werden ganze Sektoren (Blöcke) aktualisiert. Seinen Namen *Flash* (engl. Blitz) bekam dieser Speicher, weil er wesentlich schneller als EEPROM arbeitet.

Der Flash-ROM des ATmega 644 dient als Programmspeicher, in dem die kompilierten Programme abgelegt werden. Die Programmdaten werden z. B. über die JTAG-Schnittstelle oder mit Hilfe eines Bootloaders in den Flash-Speicher geschrieben und während der Programmausführung Wort für Wort daraus gelesen und ausgeführt. Der Flash-ROM kann zusätzlich zur Speicherung von Daten (etwa Texte für das LC-Display) verwendet werden.

[S]RAM

Der [S]RAM ([Static] Random Access Memory) ist flüchtiger Speicher mit wahlfreiem Zugriff. Er kann sehr schnell beliebig oft gelesen und beschrieben werden. Deshalb eignet sich seine Verwendung am besten zur Speicherung von zusätzlichen Daten, die zur Laufzeit des Programms aktiv benutzt werden. Da es sich bei RAM um einen flüchtigen Speicher handelt, gehen diese Daten nach dem Ausschalten des Mikrocontrollers verloren.

Wahlfreier Zugriff bedeutet, dass auf die Daten in beliebiger Reihenfolge lesend und schreibend zugegriffen werden kann, was den RAM zusätzlich von Flash-ROMs unterscheidet. Im Gegensatz zum Flashspeicher muss der [S]RAM nicht blockweise beschrieben werden.

Wenn zum Datenerhalt nur eine Versorgungsspannung und keine wiederholte Auffrischung benötigt wird, liegt ein SRAM vor. Das hat unter anderem den Vorteil, dass ein Mikrocontroller beliebig langsam getaktet werden kann, ohne die Daten zu verlieren.

Übersicht

Die typischen Verwendungen der einzelnen Speicherarten sind in der Tabelle 3.1 zusammengefasst.

Name	Verwendung
EEPROM	Speicher für interne Programmeinstellungen.
Flash	Programmspeicher. Wird auch zum Ablegen von Zeichenketten verwendet.
[S]RAM	Speichert zusätzliche Programmdaten zur Laufzeit.

Tabelle 3.1: Typische Verwendung der einzelnen Speicherarten.

Lernerfolgsfragen

- Was bedeutet „flüchtiger Speicher“?
- Welche Speicherarten hat ein ATmega 644? Wozu werden diese jeweils verwendet?
- Was unterscheidet EEPROM und Flash-ROM von SRAM?
- Warum eignet sich EEPROM zum Speichern von Einstellungen am besten?

3.3.3 Dynamischer Speicher

Der dynamische Speicher (engl. *Heap*) ist ein Speicherbereich im RAM, aus dem Prozesse zur Laufzeit zusammenhängende Blöcke beliebiger Größe für eigene Daten anfordern können. Wichtig ist, dass im Gegensatz zum Stack diese Blöcke in beliebiger Reihenfolge wieder freigegeben und später für einen anderen Prozess angefordert werden können. Statt lokale Variablen auf dem Stack zu verwenden, können Daten in den Heap ausgelagert werden. Diese Daten bleiben im Heap erhalten, solange der Speicherblock nicht vom Prozess selbst oder vom Betriebssystem wieder freigegeben wird.

Da Prozesse Speicherbereiche in unterschiedlichen Größen anfordern können, muss das Betriebssystem über eine Speicherverwaltung verfügen, die einzelne Allokationen und Freigaben überwacht und verarbeitet. Die Freigabe kann sowohl manuell, als auch mit Hilfe einer automatischen Speicherbereinigung (engl. *Garbage Collection*) erfolgen (z. B. sobald ein Prozess terminert). Ohne eine automatische Bereinigung kann es zu ungenutzten, besitzerlosen Speicherblöcken kommen, die von einem Prozess nicht explizit freigegeben wurden. Diese Speicherblöcke können nicht wiederverwendet werden und reduzieren die Größe des freien Speichers zusätzlich.

Allokationsstrategien

Der vom Betriebssystem zu verwaltende dynamische Speicher kann als großes Array von Bytes aufgefasst werden. Fragt ein Benutzer dynamischen Speicher an, muss das Betriebssystem in diesem Array einen Abschnitt finden, der unbelegt ist und mindestens die angeforderte Größe hat. Da hierbei der Speicher typischerweise in mehrere unterschiedlich große Stücke und Lücken unterteilt wird (*Fragmentierung*) ist das auswählen eines günstigen Abschnitts nicht trivial. Abbildung 3.2 zeigt exemplarisch einen 25 Byte großen dynamischen Speicher, in dem 9 Byte, in Form von 6 Abschnitten, belegt sind (gelb) und noch 16 Byte frei (weiß) sind.

Im Gegensatz zu Festplatten kann dynamischer Speicher nicht defragmentiert werden, da nicht in die privaten Variablen von Anwendungsprozessen eingegriffen werden kann, um dem Prozess die Adressänderung des Speicherabschnitts mitzuteilen.

Abbildung 3.2: Fragmentierter Speicher mit 25 Byte

Abbildung 3.3: Allokation von zwei Bytes mit First-Fit

Die einfachste Allokationsstrategie, *First-Fit*, beginnt die Suche nach einem passenden Abschnitt bei der ersten Adresse des dynamischen Speichers. Dabei wird die erste freie Lücke, die größer oder gleich der angeforderten Größe ist, als zu nutzender Abschnitt ausgewählt. Abbildung 3.3 zeigt den Speicher aus Abbildung 3.2 nach dem Anfordern von zwei Byte (blau).

Lernerfolgsfragen

- Welche von den im vorherigen Abschnitt vorgestellten Speicherarten wird für den dynamischen Speicher verwendet? Warum?
- Wie unterscheiden sich Heap und Stack?
- Was passiert mit einem Speicherblock, der von einem Prozess angefordert und nicht freigegeben wurde?
- Was ist Garbage Collection (GC) und wofür wird sie verwendet? Wie unterscheidet sich die hier vorgestellte GC von der in Java verwendeten GC?
- Was versteht man unter Fragmentierung? Wie entsteht Fragmentierung?
- Wie funktioniert die First-Fit Allokationsstrategie?

3.4 Hausaufgaben

Implementieren Sie die in den nächsten Abschnitten beschriebenen Funktionalitäten. Halten Sie sich an die hier verwendeten Namen und Bezeichnungen für Variablen, Funktionen und Definitionen. Beachten Sie außerdem die angegebenen Hinweise zur Implementierung.

Fangen Sie mit der Erweiterung Ihres Schedulers um die neuen Strategien an. Lesen Sie dann bitte erst den gesamten Abschnitt zu der Speicherverwaltung durch, bevor Sie

diese implementieren. Abschließend muss SPOS um die Möglichkeit der Terminierung von Anwendungsprozessen erweitert werden.

3.4.1 Implementierung der Schedulerstategien

Erweitern Sie den in Versuch 2 entwickelten Scheduler so, dass alle in Abschnitt 3.3.1 vorgestellten Strategien genutzt werden können. Bei jedem Aufruf des Schedulers muss anhand der gewählten Schedulingstrategie entschieden werden, ob der aktive Prozess fortgesetzt wird, oder ob ein anderer Prozess seine Ausführung beginnen darf.

Hinweise zur Implementierung

- Neue Strategien können in der Datei `os_scheduling_strategies.c` bzw. der zugehörigen Headerdatei angelegt werden.
- Bei der Implementierung der Schedulingstrategien können Sie davon ausgehen, dass der Prozess mit der PID (Prozess-ID) 0 - der Leerlaufprozess - immer aktiv ist und ausgewählt werden kann, falls alle anderen Prozesse nicht existieren oder nicht verfügbar sind. Stellen Sie auch sicher, den Leerlaufprozess niemals auszuwählen, wenn Anwendungsprozesse auf Ausführung warten.
- Beachten Sie, dass für einige Strategien Werte für Zeitscheibe und Alter anfallen.
- Die Auswahl des als nächstes auszuführenden Prozesses (nachdem der aktuelle Prozess terminiert ist) durch die Run-To-Completion Strategie ist beliebig.

3.4.2 Dynamischer Speicher in SPOS

Der SRAM des ATmega 644 soll im Rahmen dieses Praktikums in zwei große Bereiche aufgeteilt werden: Den dynamischen Speicher (Heap) und den Stack (dessen Verwaltung bereits in Versuch 2 vorgestellt wurde). Das Ablegen von einigen privaten Daten (wie zum Beispiel lokalen Variablen) auf dem Stack wird für alle Prozesse vom Compiler bzw. der Hardware organisiert und vollständig verwaltet. Der dynamische Speicher hingegen soll den Anwendungsprozessen einen zusätzlichen Speicher zur Verfügung stellen, welcher zur Laufzeit vom Betriebssystem verwaltet werden muss. Die dafür verwendeten Funktionen werden bewusst an die entsprechenden `stdlib.h` Funktionen angelehnt: `os_malloc` und `os_free`. Abbildung 3.4 zeigt eine erweiterte Aufteilung des SRAM aus Versuch 2. Der Heap, bestehend aus Allokationstabelle und Nutzdaten, wird im niedrigen Adressbereich des SRAMs angelegt und wächst dem Stack entgegen.

Die Allokationstabelle

Um festzuhalten, welche Speicherbereiche für welche Prozesse allokiert wurden, soll ein Teil des Speichers für eine *Allokationstabelle* (engl. *Heapmap*) reserviert werden. Diese Tabelle besteht aus Einträgen zu je vier Bit (sog. Nibbles oder Halbbytes), die den Besitzer der zu diesem Eintrag gehörenden Nutzdaten angeben. Ein Byte in der Tabelle

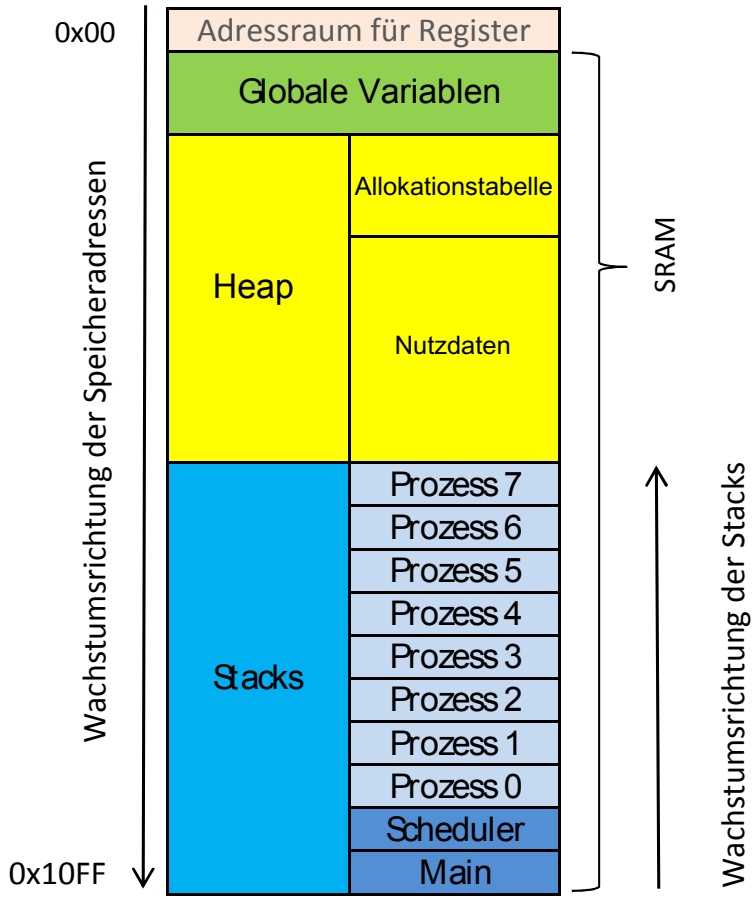

Abbildung 3.4: Datenspeicherbelegung (SRAM) des ATmega 644

verwaltet folglich zwei Bytes Nutzdaten, daher ist der Nutzdatenbereich genau doppelt so groß wie die Allokationstabelle. Je nach Belegung der vier Bit können somit jedem Byte Nutzdaten $2^{4\,\mathrm{Bit}} = 16$ verschiedene Zustände in der Heapmap zugeordnet werden. Die Zuordnung wird als „Big-Endian“ vorgenommen: Das höherwertige Halbbyte des Eintrags in der Heapmap verweist auf das Byte mit der niedrigeren Adresse. Diese Konvention erleichtert in späteren Versuchen die Ausgabe der Allokationstabelle und hilft, einzelne Speicherblöcke auf den ersten Blick zu identifizieren.

Das Beispiel in Abbildung 3.5 zeigt die Verwendung der Allokationstabelle. Das erste Byte des Speichers (Nutzdaten) ist frei, da an der entsprechenden Position eine Null in der Allokationstabelle steht. Die Werte der Allokationstabelle für die nächsten vier Bytes Nutzdaten sind 1FFF, was in diesem Protokoll bedeutet, dass diese vier Bytes dem Prozess 1 zugeordnet sind. Analog wurden die nachfolgenden zwei Bytes vom dritten Prozess reserviert (in der Allokationstabelle mit 3F markiert). Das letzte dargestellte Byte ist frei (wieder markiert durch eine 0).

Für die Zuordnung von Speicherblöcken zu Prozessen muss ein Protokoll entworfen werden, welches anhand der Werte der Allokationstabelle jedem Speicherblock eindeutig einen Prozess zuweist. Umgekehrt müssen alle von einem Prozess allokierten Speicherblöcke in der Allokationstabelle identifizierbar sein. Die Anzahl der Prozesse ist auf 7+1 (7 Anwendungs- und der Leerlaufprozess) limitiert, wobei nur die 7 Anwendungsprozesse dynamischen Speicher anfordern dürfen. Beachten Sie, dass Ihnen 12 der 16 möglichen Zustände für ein Halbbyte der Allokationstabelle zur Verfügung stehen - in Versuch 4 wird es notwendig sein, mindestens vier weitere Zustände pro Speicherblock vergeben zu können!

Für diese Aufgabe können Sie sich an dem hier beschriebenen Verfahren orientieren oder Ihr eigenes Protokoll entwerfen. Beachten Sie jedoch, dass die Vergabe eines Speicherbereiches beliebiger Länge unterstützt werden muss (falls genügend freier Speicher zur Verfügung steht) - insbesondere Speicherblöcke der Länge 1 Byte, aber auch Blöcke mit einer Länge > 255 Bytes. Insbesondere muss es möglich sein, $\lfloor n/l \rfloor$ Speicherabschnitte der Länge l Byte (durch den gleichen Prozess) anzufordern, falls der nutzbare Speicher die Größe n Byte hat.

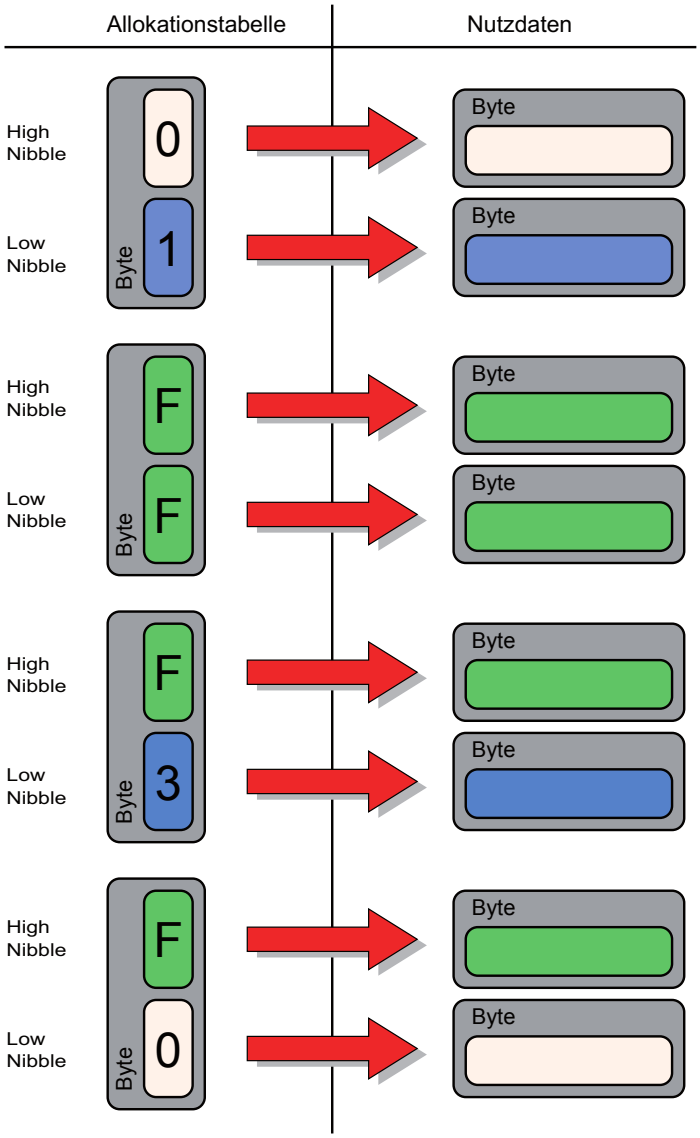

Abbildung 3.5: Verdeutlichung des Zusammenhangs zwischen Allokationstabelle und Nutzdaten

Hinweis

- Überlegen Sie sich den Aufbau des Protokolls, bevor Sie es implementieren!
- Der Leerlaufprozess wird niemals Speicher anfordern.
- Es muss möglich sein Speicherblöcke der Länge 1 Byte oder mit einer Länge > 255 Bytes zu allokieren.
- Es muss möglich sein, einem Speicherbereich noch mindestens 4 weitere Zustände zuzuordnen (abgesehen von den bis zu 7 Zuständen, die jeweils einem bestimmten Besitzerprozess entsprechen).
- Überprüfen Sie ob Ihr Protokoll diesen Anforderungen genügt, falls Sie ein von obigem Vorschlag abweichendes Protokoll entwerfen.

Wie in Abschnitt 3.3.2 bereits erwähnt, stehen dem ATmega 644 insgesamt 4096 Bytes Arbeitsspeicher (SRAM) zur Verfügung. Diese sollen gleichmäßig auf dynamischen Speicher und Stack aufgeteilt werden. Der Mikrocontroller legt verschiedene Daten (z. B. globale Variablen des Betriebssystems) automatisch am Anfang des SRAM ab. Daher sollte zum Anfang des Heap ein Sicherheitsabstand gewährt werden. Teilen Sie den restlichen verfügbaren Speicher im Verhältnis 1:2 in die Allokationstabelle und den nutzbaren Datenbereich auf.

Die bereits im SRAM verbrauchte Größe kann nach der Kompilierung des Projekts im AVR Studio abgelesen werden: Im *Build*-Fenster steht beispielsweise

```
Data:          83 bytes (2.0% Full)
(.data + .bss + .noinit)
```

In diesem Fall würde ein Abstand von 100 Bytes ausreichen (83 Bytes + Sicherheitsabstand). Abhängig von Ihrer Implementierung muss dieser Wert angepasst werden.

Achtung

- Berücksichtigen Sie, dass Sie diesen Wert ggf. in den nächsten Versuchen anpassen müssen, wenn Sie das Betriebssystem erweitern und z. B. neue globale Variablen anlegen.
- Bedenken Sie, dass der für Sie nutzbare Datenbereich nicht bei Adresse 0 beginnt! Die Startadresse ist im Abschnitt 6.2: *SRAM Data Memory* des Datenblattes für ATmega 644 dokumentiert.
- Achten Sie darauf, dass sämtliche Zeichenketten, wie im *Begleitenden Dokument zum Praktikum Systemprogrammierung* beschrieben im Flash-Speicher abgelegt werden!

Dieser Wert kann exakt zur Laufzeit ermittelt werden, da er nach dem Kompilieren von SPOS feststeht und durch den Linker durch das Symbol `__heap_start` markiert wird. Durch Auslesen der Adresse dieser Variable lässt sich also zur Laufzeit die Größe des Abschnitts der globalen Variablen bestimmen. Dies erlaubt eine Sicherheitsüberprüfung, ob der zur Designzeit durch das Define `HEAPOFFSET` (siehe nachfolgenden Abschnitt) festgelegte Anfang des Heaps die globalen Variablen überschreibt.

Diese Prüfung ist für Sie optional, jedoch sehr empfohlen, da Fehler die durch das Überschreiben globaler Variablen entstehen, äußerst schwer zu finden sind.

Lernerfolgsfragen

- Wieso ist es sinnvoll festzuhalten, welcher Speicherbereich reserviert wurde?
- Wieso ist es sinnvoll festzuhalten, welcher Prozess einen bestimmten Speicherbereich belegt?
- Welches Verhalten sieht Ihr Protokoll vor, falls derselbe Prozess zwei Speicherbereiche der Länge 1 anfordert, und diese direkt nebeneinander abgelegt werden? Kann es bei der Freigabe zur Verwechslung mit einem Speicherblock der Länge 2 kommen?
- Sieht die Spezifikation Ihres Protokolls vor, dass ein allokierter Speicherblock freigegeben werden kann, wenn die übergebene Adresse nicht auf das erste Byte dieses Blocks zeigt? Falls nein, wie kann dies ermöglicht werden?
- Was kann geschehen, wenn Sie den Sicherheitsabstand zum Bereich der globalen Variablen zu gering bzw. zu groß kalkulieren?
- Wieso kann die Grösse der globalen Variablen nicht durch ein Define zur Compilezeit festgelegt werden, sondern steht erst zur Linkzeit fest?
- Ab welcher Adresse beginnend wird der SRAM adressiert? Was liegt davor?

Konstanten festlegen

Ermitteln Sie alle benötigten Informationen über den Speicher und fügen Sie diese dem Header `defines.h` hinzu. Dazu sollten mindestens gehören:

- Größe des Sicherheitsabstands
- Startadresse der Allokationstabelle
- Größe der Allokationstabelle
- Startadresse des Speichers für Nutzdaten
- Größe des Speichers für Nutzdaten
- Endadresse des Speichers für Nutzdaten

Dabei ist mit Endadresse stets die erste ungülte Adresse *nach* dem jeweiligen Speicherabschnitt gemeint.

Verwenden Sie dafür das `#define`-Schlüsselwort, wie im *Begleitenden Dokument zum Praktikum Systemprogrammierung* beschrieben, sowie die Bezeichner im Listing 3.1.

```
1 #define HEAPOFFSET          ?
2 #define HEAP_MAP_SIZE       ?
3 #define HEAP_USE_SIZE       ?
4 #define HEAP_MAP_START      ?
5 #define HEAP_USE_START      ?
6 #define HEAP_MAP_END        ?
7 #define HEAP_USE_END        ?
```

Listing 3.1: Notwendige Definitionen für den dynamischen Speicher.

Aufteilung in Schichten

Die zu implementierende Speicherverwaltung soll soweit abstrakt sein, dass der Zugriff auf das eigentliche Speichermedium – in diesem Versuch auf den SRAM – für die Anwendungsprozesse transparent bleibt und das Medium beliebig ausgetauscht werden kann. Zu diesem Zweck wird die Speicherverwaltung in zwei Schichten unterteilt: Die untere Schicht stellt einen Treiber für direkte Zugriffe auf das Speichermedium bereit, die obere Schicht (wie in den vorherigen Abschnitten beschrieben) bearbeitet die Speicheranfragen der Anwendungsprozesse auf Basis des Treibers. Dieses Treibermodell wird eingeführt, da in späteren Versuchen andere Speichermedien als der SRAM genutzt werden.

Speichertreiber sind Instanzen der unteren Schicht und enthalten Informationen über das benutze Speichermedium. Dazu gehören unter anderem seine Größe, verfügbarer Adressraum und Funktionen zur Initialisierung des Mediums sowie zum direkten Schreiben und Lesen der Daten. Für jedes benutzte Speichermedium muss ein eigener Treiber angelegt werden. Diese Treiber werden als Parameter in die Funktionen der oberen Schicht übergeben und ermöglichen den Zugriff auf das jeweilige Medium. Abbildung 3.6 verdeutlicht diese Aufteilung.

Untere Schicht: Treiber

Der Speichertreiber wird als eine Struktur realisiert, die alle charakteristischen Werte des Mediums, sowie die nötigen Routinen (bzw. Funktionszeiger darauf) enthält. Diese Struktur stellt eine abstrakte Basisklasse dar, die für jedes konkrete Speichermedium instanziiert und initialisiert werden muss.

Hinweis

Für eine bessere Lesbarkeit des erzeugten Codes ist es hilfreich, die Typen der nötigen Funktionszeiger vorher in `os_mem_drivers.h` festzulegen (vgl. Doxygen-Dokumentation).
Legen Sie dort auch alle speicherbezogenen Typen an, die im Folgenden benötigt werden.

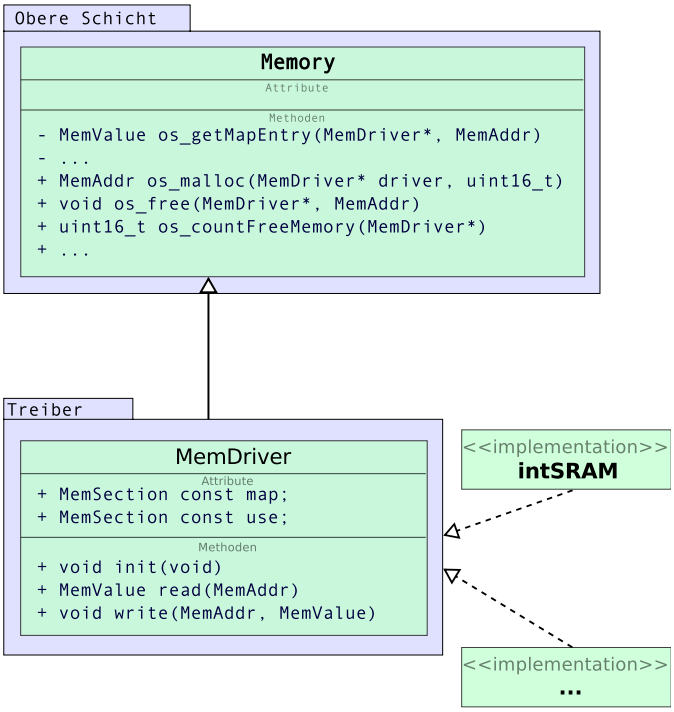

Abbildung 3.6: Zusammenspiel zwischen oberer und unterer Schicht.

Üblicherweise erfolgt der Zugriff auf die Daten im SRAM über `uint8_t*` Zeiger. Ein solcher Zeiger enthält eine 16-Bit lange Adresse und wird für jede Lese-/Schreiboperationen dereferenziert. Im Rahmen des Praktikums müssen jedoch alle Operationen auf dem Speicher nur indirekt über den jeweiligen Treiber erfolgen. Daher werden statt Zeigern Variablen des Typs `uint16_t` (oder `uintptr_t`) verwendet, wodurch keine direkte Dereferenzierung mehr möglich ist. Für eine bessere Lesbarkeit wird der neue Datentyp `MemAddr` angelegt, der intern mit Hilfe des Schlüsselworts `typedef` als `uint16_t` definiert ist. Somit ist im Quelltext sofort ersichtlich, dass es sich um Speicheradressen handelt. Außerdem kann in späteren Versuchen damit ein externes Speichermedium adressiert werden, das 13 Bit lange Adressen ohne Bezug zum SRAM verwendet. Analog wird der Typ `MemValue` für einzelne durch `MemAddr` referenzierten *Speicheratome* verwendet. Als Atome werden in diesem Kontext einzelne Byte bezeichnet, da diese nicht weiter auf verschiedene Benutzer verteilt werden können. Folglich hat `MemValue` die Größe eines Bytes, kann also als `uint8_t` definiert werden.

Legen Sie in der Datei `os_mem_driver.h` die Struktur `MemDriver` des allgemeinen Speichertreibers an. Definieren Sie den Typ `MemDriver` als eine Kurzform für `struct MemDriver`. Fügen Sie der angelegten Struktur Konstanten für alle charakteristischen Werte des benutzten Speichermediums, sowie Funktionszeiger für die folgenden Zugriffsroutinen hinzu:

- Eine Routine `void init(void)` zur Initialisierung des Speichermediums. Hier soll

unter anderem die Allokationstabelle vorbereitet werden.

- Eine Leseroutine `MemValue read(MemAddr addr)`, die den Wert an der gegeben Adresse `addr` ausliest und zurückgibt.
- Eine Routine `void write(MemAddr addr, MemValue value)` zum Schreiben auf das Medium. Sie speichert den Wert `value` an die Adresse `addr`.

Weitere Informationen zu den einzelnen Komponenten und deren Verwendung können in der Doxygen-Dokumentation zu diesem Versuch gefunden werden.

Treiber für den internen SRAM

Erstellen Sie die Datei `os_mem_drivers.c`, in der der Treiber für den internen SRAM angelegt wird. Implementieren Sie in `os_mem_drivers.c` die im letzten Abschnitt angegebenen Funktionen für den direkten Zugriff auf den SRAM. Achten Sie dabei auf korrekte Parameter und Rückgabewerte.

Definieren Sie eine Instanz der Schnittstelle mit dem Bezeichner `intSRAM` und dem Typ `MemDriver*`, über die auf den SRAM zugegriffen werden kann. Initialisieren Sie die Funktionspointer dieser Schnittstelle (vlg. Abbildung 3.6) mit den zuvor in `os_mem_drivers.c` implementierten Routinen und legen Sie die wichtigen Konstanten der Allokationstabelle und des Heaps an. Weisen Sie diesen Konstanten die Werte entsprechender Definitionen zu, die im Abschnitt 3.4.2 erläutert wurden.

Rufen Sie die Funktion `init` der Treiberstruktur an einer geeigneten Stelle in Ihrem Code auf, damit die Allokationstabelle des Heaps beim Starten des Betriebssystems initialisiert werden kann.

Hinweis

- Attribute einer Treiberstruktur, die mit dem Schlüsselwort `const` deklariert wurden, können zur Laufzeit nicht initialisiert werden. Ein Beispiel der korrekten Vorgehensweise kann in Abschnitt *Umgang mit Daten* des *Begleitenden Dokuments zum Praktikum Systemprogrammierung* gefunden werden.
- Da der Zugriff auf den internen SRAM nur über den Treiber ermöglicht werden soll, werden die Zugriffsfunktionen *private* deklariert. In diesem Kontext bedeutet das, dass die Funktionsköpfe nicht in der Headerdatei veröffentlicht werden und somit verdeckt für die anderen Module bleiben.
- Beachten Sie bei der Implementierung der Treiberstruktur und der Adresslogik folgende Abschnitte des *Begleitenden Dokuments zum Praktikum Systemprogrammierung*: *Lesbares Bitshifting*, *Typecasts* und *Eigene Datentypen erstellen*.
- Wie Sie die `MemDriver` Struktur genau aufbauen bleibt Ihnen überlassen. Allerdings müssen Anwendungsprogramme direkt auf die Schreib- und Lesefunktionen zugreifen, sodass der Zeiger der `MemDriver` Struktur der auf die Schreibfunktion zeigt, zwingend den Bezeichner `write` haben muss. Der Zeiger auf die Lesefunktion muss aus dem gleichen Grund den Bezeichner `read` haben.

Obere Schicht: Speicherverwaltung

Erweitern Sie Ihr Projekt mit der neuen Quelldatei `os_memory.c` und einem passenden Header. Diese Dateien sollen dem Betriebssystem Funktionen zur Allokation und Freigabe des Speichers auf dem Heap zur Verfügung stellen.

Achtung

Folgende Funktionen der Speicherverwaltung stellen kritische Bereiche dar und müssen atomar bleiben.

- **`MemAddr os_malloc(MemDriver* driver, uint16_t size)`**:
  Wird diese Allokierungsfunktion von einem Prozess aufgerufen, sollen so viele zusammenhängende Bytes mithilfe des übergebenen Zeigers auf dem Speichertreiber `driver` allokiert werden wie der Parameter `size` es angibt. D.h. die Allokationstabelle muss entsprechend Ihrem Protokoll aktualisiert werden. Anschließend soll die Adresse des ersten Bytes des gefundenen Speicherbereichs an den Prozess zurückgegeben werden.

Das tatsächliche Auswählen des jeweiligen Speicherbereichs soll dabei analog zu den Schedulingstrategien austauschbar sein und von Allokationsstrategien vorgenommen werden. Implementieren Sie für diesen Versuch die *First-Fit* Strategie gemäß Abschnitt 3.3.3.

- **`void os_free(MemDriver* driver, MemAddr ptr)`**:
  Diese Funktion soll den Speicherbereich, auf den der übergebene Zeiger verweist, freigeben. Alle entsprechenden Einträge in der Allokationstabelle müssen folglich auf 0 gesetzt werden. Beachten Sie, dass der von `os_malloc` reservierte Speicher privat ist und dem Prozess gehört, der den Speicher angefordert hat. Ein Speicherbereich darf nur von seinem Besitzer wieder freigegeben werden. Weiterhin kann der übergebene Zeiger innerhab der Grenzen des Bereichs auf den er zeigt verschoben sein, anstatt auf das erste Bytes des Blocks zu zeigen. Auch in diesem Fall muss der *gesamte* Speicherbereich korrekt freigegeben werden.

- **`uint16_t os_getDriverMapSize(MemDriver const* driver)`**,
  `uint16_t os_getDriverUseSize(MemDriver const* driver)`,
  `MemAddr os_getDriverMapStart(MemDriver const* driver)`,
  `MemAddr os_getDriverUseStart(MemDriver const* driver)`,
  `MemAddr os_getDriverMapEnd(MemDriver const* driver)` und
  `MemAddr os_getDriverUseEnd(MemDriver const* driver)`:
  Diese Funktionen dienen zum Zugriff auf die von Ihnen spezifizierte Struktur `MemDriver`. Sie werden insbesondere vom Taskmanager verwendet und sollen jeweils die Grenzen der Allokationstabelle (...`Map`...) bzw. des Nutzdatenbereichs (...`Use`...) zurückliefern.

  Hierbei ist mit ...`Size` jeweils die größe des jeweiligen Bereichs (Allokationstabelle oder Nutzdaten) in Byte gemeint. Die ...`Start` Funktionen liefern jeweils die erste Adresse des jeweiligen Bereichs zurück, während die auf ...`End` endenden Funktionen die erste ungültige Adresse (das Ende) des jeweiligen Bereiches zurückgeben.

- **`uint16_t os_getChunkSize(MemDriver const* driver, MemAddr addr)`**:
  Liefert die Größe in Bytes eines belegten Abschnitts (*Chunk*) zurück. Falls der Speicherabschnitt frei ist, muss 0 zurückgeliefert werden.

In der Doxygen-Dokumentation zu diesem Versuch finden Sie einige weitere Funktionsprototypen mit detaillierten Erklärungen, die zur Umsetzung dieser Aufgabe hilfreich sind. Funktionen, die in diesen Versuchsunterlagen nicht explizit gefordert werden, müssen von Ihnen nicht implementiert werden. Es wird jedoch stark dazu geraten, für den Zugriff auf die Allokationstabelle, analog zur Doxygen-Dokumentation, mehrere Hilfsfunktionen zu implementieren.

Hinweis

- Alle Fehler und potentielle Gefahren sollen explizit ausgegeben werden.
- Bauen Sie an sinnvollen Stellen Parameterüberprüfungen ein. Dabei soll insbesondere auf die Einhaltung der Speichergrenzen geachtet werden. Die charakteristischen Werte können immer der übergebenen Treiberstruktur entnommen werden.
- Falls die Anfrage nach einem Speicherbereich von `os_malloc` nicht bedient werden kann, da nicht mehr ausreichend Speicher zur Verfügung steht, oder weil der angeforderte Bereich grundsätzlich nicht in den Speicher passt, ist dies **kein** kritischer Fehler. Daher muss `os_malloc` in diesem Fall `NULL` zurückliefern.
- In der Präsenzaufgabe zu diesem Versuch werden Sie mindestens eine weitere Allokationsstrategie implementieren und in Versuch 5 kommen weitere dazu. Beachten Sie bereits jetzt diese Austauschbarkeit der Allokationsstrategien und treffen Sie die nötigen Vorbereitungen.
- Überprüfen Sie, ob Ihre Implementierung bei Bedarf sicherstellt, dass eine atomare Abarbeitung gewährleistet ist.

Lernerfolgsfragen

- In welcher Form muss die Allokationstabelle initialisiert werden, damit die Speicherverwaltung korrekt funktioniert?
- Müssen für die korrekte Funktionsweise der Speicherverwaltung auch die Nutzdaten initialisiert werden? Warum (nicht)?
- Wie viel Speicher kann mit Ihrer Implementierung von `os_malloc` maximal angefordert werden?
- Welcher Datentyp wird in SPOS für Speicheradressen verwendet?
- Was ist die Aufgabe der einzelnen Schichten der Speicherverwaltung? Warum ist diese Aufteilung sinnvoll?
- Welche Prozesse dürfen einen Speicherbereich an der Adresse `addr` mit `os_free(..., addr)` freigeben?

3.4.3 Terminierung von Prozessen

Um den Verwaltungsaufwand möglichst gering zu halten, wurde bis jetzt angenommen, dass die laufenden Prozesse nicht terminieren können. Das ist im Allgemeinen eine starke Vereinfachung. Um den Anwendungsprozessen die Terminierung zu ermöglichen, sollen diese in einer Funktion gekapselt werden.

Legen Sie in der Datei `os_scheduler.c` die Funktion `void os_dispatcher(void)` an, die die Kapselung der Anwendungsfunktion übernehmen soll. Der Ablauf der Ausführung ändert sich wie folgt:

1. In der Funktion `os_exec` wird nicht mehr das eigentliche Programm gestartet, sondern immer die Funktion `os_dispatcher`.
2. In der Funktion `os_dispatcher` wird die PID i des aktiven Prozesses, sowie die zugehörige Programmfunktion j bestimmt.
3. Aus `os_dispatcher` wird die Programmfunktion mittels des Funktionszeigers direkt aufgerufen.
4. Sobald die Programmfunktion abgearbeitet und wieder verlassen wurde, kehrt die Ausführung in die Funktion `os_dispatcher` zurück.
5. Da die Funktion verlassen wurde, muss der aktive Prozess terminiert werden. Dazu wird das Array `os_processes` aufgeräumt und der Slot des aktiven Prozesses freigegeben.
6. Garbage Collection: Alle von diesem Prozess angeforderten Speicherblöcke werden freigegeben (Präsenzaufgabe).
7. Im letzten Schritt fällt die Funktion `os_dispatcher` in eine Endlosschleife und wartet, bis der Scheduler den nächsten Prozess auswählt.

Die Funktion `os_dispatcher` startet also keinen neuen Prozess, sondern führt innerhalb des aktiven Prozesses lediglich die entsprechende Programmfunktion aus. Der Prozess selber wird nach wie vor in `os_exec` erzeugt.

> **Hinweis**
>
> Überlegen Sie sich, welche Codesegmente in Ihrer Implementierung von `os_dispatcher` auf keinen Fall vom Scheduler unterbrochen werden dürfen und an welchen Stellen eine Unterbrechung erlaubt sein muss.

Zum vorzeititgen Terminieren von Prozessen, etwa aus dem Taskmanager heraus oder durch andere Prozesse, muss zusätzlich die Funktion `bool os_kill(ProcessID pid)` implementiert werden, welche sich stark mit `os_dispatcher` überschneidet. Es ist daher überlegenswert, ob `os_dispatcher` in geeigneter Form `os_kill` aufruft.

Lernerfolgsfragen

- Können sich zwei Prozesse gegenseitig stören, wenn sie sich gleichzeitig in derselben Funktion befinden? Ist die Konsistenz lokaler (globaler) Variablen betroffen?
- Aus welchem Grund darf der Leerlaufprozess niemals beendet werden?
- Kann es zu Problemen kommen, wenn der Slot des aktiven Prozesses außerhalb (innerhalb) des Schedulers freigegeben wird?
- Müssen die Interrupts ein- oder ausgeschaltet sein, wenn die Endlosschleife betreten wird? Warum (nicht)?
- Was geschieht wenn ein Prozess `os_kill` mit seiner eigenen PID aufruft?
- Was geschieht wenn ein Prozess terminiert, der sich in einer kritischen Sektion befindet?

3.4.4 Zusammenfassung

Die zu spezifizierenden Typen und Funktionen im Überblick:

- Datentypen
  - `MemAddr` (16 Bit Ganzzahl, vorzeichenlos)
  - `MemValue` (8 Bit Ganzzahl, vorzeichenlos)
  - `struct MemDriver`
  - `enum AllocStrategy`
  - Für alle `enum`s, `struct`s und `union`s passende `typedefs`
- Defines
  - `HEAPOFFSET`, `HEAP_{MAP,USE}_{SIZE,START,END}`
  - `RAM_DRIVER_LIST_LENGTH` (derzeit gibt es nur einen einzigen Speicher-Treiber, daher 1)
  - `ALLOC_STRATEGY_COUNT` (derzeit ist nur nur eine Allokationsstrategie implementiert, daher 1)
- Globale Variablen
  - `MemDriver* const intSRAM` (darf auch ein Define sein)
- Funktionen

  - `void os_setSchedulingStrategy(SchedulingStrategy)`
  - `SchedulingStrategy os_getSchedulingStrategy(void)`
  - Die Methoden des `MemDriver` Typs.
  - `MemAddr os_malloc(MemDriver*, uint16_t)`
  - `void os_free(MemDriver*, MemAddr)`
  - `uint16_t os_getDriver{Map,Use}Size(MemDriver const*)`
  - `MemAddr os_getDriver{Map,Use}{Start,End}(MemDriver const*)`
  - `uint16_t os_getChunkSize(MemDriver const* driver, MemAddr addr)`
  - `MemDriver* os_lookupMemDriver(uint8_t)` (gibt `intSRAM` zurück falls 0 übergeben wurde, sonst `NULL`)
  - `void os_setAllocationStrategy(MemDriver*,AllocStrategy)` (setzt die Allokationsstrategie)
  - `AllocStrategy os_getAllocationStrategy(MemDriver*)` (gibt die Allokationsstrategie zurück)
  - `void os_dispatcher(void)`
  - `void os_kill(ProcessID)`

- Weitere Funktionalität
  - Schedulingstrategien: Run-To-Completion, RoundRobin, InactiveAging
  - Allokationsstrategie: First-Fit

3.5 Präsenzaufgaben

Sie erhalten von den Betreuern während des Versuchs eine Sammlung von Testprogrammen, mit denen die Funktionalität Ihres Betriebssystems zum aktuellen Entwicklungsstand getestet werden kann. Wenn Ihre eigene Anwendungsprogramme fehlerfrei unterstützt werden, testen Sie diese Referenzprogramme. Wenn es mit diesen Probleme gibt, haben Sie wahrscheinlich bestimmte Anforderungen nicht erfüllt oder gewisse Sonderfälle nicht abgefangen. Ergänzen Sie Ihr Projekt entsprechend. Die Implementierungshinweise und Lernerfolgsfragen weisen meist auf notwendige Kriterien für den erfolgreichen Durchlauf der Testprogramme hin.

3.5.1 Garbage Collection

Implementieren Sie die Funktion

```
void os_freeProcessMemory(MemDriver*, ProcessID)
```

die den gesamten Speicher freigibt, der von dem Prozess mit der angegebenen PID über den angegebenen Treiber allokiert wurde.

Fügen Sie in der Funktion `os_dispatcher` bzw. `os_kill` einen Aufruf dieser Funktion ein, um den Speicher von gerade beendeten Prozessen automatisch freizugeben.

3.5.2 Allokationsstrategie (optional)

Implementieren Sie eine zusätzliche Allokationsstrategie, anhand welcher der Speicherbereich in `os_malloc` ausgewählt wird. Mögliche Strategien werden Ihnen von einem Betreuer zugeteilt und erklärt.

3.6 Pinbelegungen

Port	Pin	Belegung
PORTA	1	frei
	2	LCD Pin 6
	3	LCD Pin 5
	4	LCD Pin 4
	5	LCD Pin 3
	6	LCD Pin 2
	7	LCD Pin 1
	8	LCD Pin 0
PORTB	1	frei
	2	frei
	3	frei
	4	frei
	5	frei
	6	frei
	7	frei
	8	frei
PORTC	1	Button 1
	2	Button 2
	3	Reserviert für JTAG
	4	Reserviert für JTAG
	5	Reserviert für JTAG
	6	Reserviert für JTAG
	7	Button 3
	8	Button 4
PORTD	1	frei
	2	frei
	3	frei
	4	frei
	5	frei
	6	frei
	7	frei
	8	frei

Die vorgegebene Pinbelegung des ATmega 644 in Versuch 3.