RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHRSTUHL
INFORMATIK 11
SOFTWARE FÜR
EINGEBETTETE
SYSTEME

Praktikum Systemprogrammierung

Versuch 4

Gemeinsamer Speicher

Lehrstuhl für Informatik 11 - RWTH Aachen

16. Mai 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

4 Gemeinsamer Speicher

4.1 Versuchsinhalte

Die Interprozesskommunikation ist ein wichtiger Bestandteil von Betriebssystemen. Um Prozessen die Möglichkeit zu geben, untereinander Daten auszutauschen, wird eine gemeinsame Kommunikationsschnittstelle benötigt. Eine mögliche Methode hierfür stellen die in diesem Versuch behandelten gemeinsam genutzten Speicherbereiche dar.

Um bei konkurrierenden Zugriffen eine korrekte Verwaltung von Ressourcen sicherzustellen, ist es notwendig, Methoden zur Zugriffssynchronisation einzuführen. Hierbei sind verschiedene Realisierungen denkbar. Einige davon werden in diesem Versuch vorgestellt und schließlich implementiert.

Weiterhin werden in diesem Versuch die Allokationsstrategien aus Versuch 3 (*Heap / Schedulingstrategien*) um weitere Strategien ergänzt.

4.2 Lernziel

Das Lernziel dieses Versuchs ist das Verständnis der folgenden Zusammenhänge:

- Konzepte des gemeinsamen Speichers
- Kontrolle und Auflösung von Speicherabhängigkeiten
- Blockierte Prozesse und deren Behandlung
- Weitere Strategien zur Allokation von Speicher

4.3 Grundlagen

In diesem Versuch wird das Konzept der Speicherverwaltung aus Versuch 3 (*Heap / Schedulingstrategien*) erweitert. Prozessen soll die Möglichkeit gegeben werden, gemeinsamen Speicher allozieren zu können und somit Daten auszutauschen. Auf die daraus resultierenden Zugriffskonflikte bei gleichzeitigem Zugriff wird im folgenden Kapitel eingegangen.

Im Anschluss daran wird erläutert, wie man Prozesse, die durch gleichzeitigen Zugriff auf gemeinsamen Speicher blockiert wurden, ressourcenschonend verwalten kann.

Zuletzt werden neue Strategien zur Allokation von Speicherbereichen des Heaps vorgestellt.

4.3.1 Gemeinsamer Speicher

Gemeinsamer Speicher bezeichnet einen allozierten Speicherbereich, der im Gegensatz zu privatem Speicher von jedem Prozess gelesen oder beschrieben werden darf. Ist das Scheduling präemptiv, d. h. werden Prozesse unerwartet unterbrochen, treten unweigerlich Konflikte auf, die behandelt werden müssen:

- *Liest* ein Prozess i aus einem gemeinsamem Speicherbereich und wird unterbrochen, so würde eine *Schreiboperation* von Prozess j auf denselben Speicherbereich das dortige Datum so verändern, dass Prozess i ein undefiniertes Datum aus dem Speicherbereich liest. Der Lesevorgang muss also vor dem Beginn des Schreibvorgangs abgeschlossen werden. Diese Abhängigkeit wird mit *read-before-write* bezeichnet.
- *Schreibt* ein Prozess i in einen gemeinsamen Speicherbereich und wird unterbrochen, läuft ein Prozess j, der aus diesem Speicherbereich *liest*, Gefahr, ein undefiniertes Datum zu lesen, da Prozess i nur einen Teil des Speicherbereichs überschrieben hat. Der Schreibvorgang muss also vor dem Beginn des Lesevorgangs abgeschlossen werden. Diese Abhängigkeit wird mit *write-before-read* bezeichnet.
- Im letzten betrachteten Konflikt *schreibt* ein Prozess i in einen gemeinsamen Speicherbereich und wird von einem Prozess j unterbrochen, der in diesen Speicherbereich *schreibt*. Während der beiden Schreiboperationen und ggf. auch danach ist der Inhalt des Speicherbereiches undefiniert. Der zweite Schreibvorgang darf also erst starten, nachdem der erste Schreibvorgang abgeschlossen wurde. Diese Abhängigkeit wird mit *write-before-write* bezeichnet.

Das Betriebssystem muss sicherstellen, dass eine Schreiboperation nicht unterbrochen wird bzw. keine andere Operation unterbricht. Dagegen soll aber das mehrfache Lesen eines Bereiches erlaubt werden. Dies soll für jeden Speicherbereich individuell festgehalten werden. Damit Speicherbereiche zur gemeinsamen Nutzung markiert werden können, muss das Protokoll der Allokationstabelle erweitert werden.

Auf der Programmierebene erfordert die Verwaltung des gemeinsamen Speichers neue Funktionen. Abgeleitet von der Bezeichnung `shared heap` für gemeinsamen Speicher werden diese mit dem Zusatz `sh` im Funktionsnamen kenntlich gemacht.

Es müssen Varianten von `os_malloc` und `os_free` implementiert werden, mit denen gemeinsame Speicherbereiche alloziert oder freigegeben werden können. Diese heißen dementsprechend `os_sh_malloc` und `os_sh_free`. Darüber hinaus werden Funktionen implementiert, die den Schreib- und Lesezugriff auf diese Variablen koordinieren und somit die zuvor beschriebenen Konflikte verhindern. Diese Funktionen heißen `os_sh_read` und `os_sh_write`.

Der Zugriff auf gemeinsamen Speicher erfolgt indirekt. Dies geschieht durch das Kopieren von Daten in bzw. aus anderen Speicherbereichen, auf die der Prozess Zugriff hat. Diese Speicherbereiche können globale oder lokale Variablen sowie private Speicherbereiche auf dem Heap sein. In Listing 4.1 ist ein konkretes Zugriffsbeispiel für gemeinsamen Speicher zu finden.

Lernerfolgsfragen

- Warum ist es notwendig, die gemeinsam genutzten Speicherbereiche zum Schreiben bzw. Lesen zu öffnen und zu schließen? Warum sollte nicht direkt auf die Speicherstellen zugegriffen werden, wie bei den privaten Speicherbereichen? Können Sie ein Beispiel angeben?
- Warum stellt es keinen Konflikt dar, wenn mehrere Leseoperationen gleichzeitig durchgeführt werden sollen? Überlegen Sie, was die Anforderung von mindestens zwei gleichzeitigen Lesezugriffen für die Erweiterung Ihres Protokolls bedeutet.

4.3.2 Blockierte Prozesse

Zur Lösung des Problems der Schreib- und Lesekonflikte werden Synchronisationsmechanismen benötigt, die bestimmte Zugriffe auf einen Speicherbereich zeitweise nur durch einen Prozess i zulassen. Eine erste naive Lösung wäre, einen weiteren Prozess j, der während des Zugriffs von i auf denselben Speicherbereich zugreifen möchte, in einer Schleife aktiv warten zu lassen, bis i den Speicherbereich wieder freigibt. Dieser Weg verbraucht jedoch unnötig Prozessorzeit.

Eine bessere Lösung der aufgezeigten Problematik ist, dass der blockierte Prozess dem Betriebssystem mitteilt, dass er auf die ihm noch zustehende Restzeit verzichtet. Das Betriebssystem entzieht diesem Prozess daraufhin *sofort* die Kontrolle und verteilt die Rechenkapazität neu. An dieser Stelle darf nicht auf den nächsten Aufruf des Schedulers durch den Timerinterrupt gewartet werden.

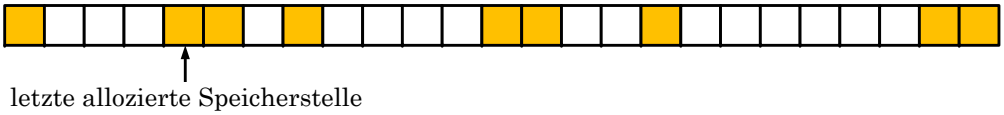
Lernerfolgsfragen

- Wann ist es für einen Prozess sinnvoll, die gesamte ihm noch zugeteilte Rechenzeit abzugeben?
- Wie könnte ein konkretes Beispiel für einen blockierten Prozess aussehen?
- Auf welche Weise können Sie bewirken, dass das Betriebsmittel Prozessor unmittelbar freigegeben wird?

4.3.3 Allokationsstrategien

Zusätzlich zu der in Versuch 3 (*Heap / Schedulingstrategien*) implementierten Allokationsstrategie *First-Fit* und gegebenenfalls weiteren optional implementierten Strategien sollen nun weitere ergänzt werden. Diese werden im Folgenden vorgestellt und zusätzlich anhand eines Beispiels veranschaulicht.

Die Speicherbelegung vor der Allokation ist in der folgenden Abbildung dargestellt. Gelbe Zellen sind belegt und weiße Zellen sind noch frei. Der Pfeil markiert das Ergebnis der letzten Allokation. Jeder der folgenden Strategien wird beispielhaft mit der Allokation von zwei Byte erläutert.



- *Next Fit* bzw. *Rotating First Fit*
  Es wird, analog zu *First Fit*, der erste passende Speicherbereich gesucht. Die Suche beginnt jeweils bei der ersten Adresse nach dem zuletzt zugewiesenen Speicherbereich. Wenn also zuletzt ein Speicherbereich der Größe k an Adresse p vergeben wurde, wird die Suche an der Adresse $p + k$ begonnen (unter der Voraussetzung, dass dies eine gültige Adresse ist, ansonsten wird am Anfang des Speichers begonnen).

  Hierbei ist der Fall zu berücksichtigen, dass, wenn man am Ende des Speichers angekommen ist, es u.U. Bereiche am Anfang gibt, die noch nicht untersucht wurden.

  Folgende Abbildung zeigt exemplarisch die Speicherbelegung nach Anwendung dieser Strategie.

- *Best Fit*
  Es wird der kleinste passende Speicherbereich genutzt. Dazu muss für jede Anfrage der gesamte Speicherbereich, beginnend bei der ersten Adresse, durchsucht werden, es sei denn, es wird ein Speicherbereich gefunden, der exakt passt. In diesem Fall wird die Suche abgebrochen und der gefundene Speicherbereich genutzt. Folgende Abbildung zeigt exemplarisch die Speicherbelegung nach Anwendung dieser Strategie.

- *Worst Fit*
  Es wird der größte passende Speicherbereich ausgewählt. Auch hier wird mit der ersten Adresse begonnen. Folgende Abbildung zeigt exemplarisch die Speicherbelegung nach Anwendung dieser Strategie.

Lernerfolgsfragen

- In welchem Fall kann bei der *Worst Fit* Strategie die Suche vorzeitig abgebrochen werden?
- Kann die *Next Fit* Strategie mit Hilfe der *First Fit* Strategie implementiert werden? Wie oft würde der Speicher im schlechtesten Fall nach einer freien Speicherstelle durchsucht?

4.4 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.

4.4.1 Gemeinsamer Speicher

Der Datenaustausch zwischen den Prozessen soll über gemeinsam genutzte Speicherbereiche erfolgen. Dazu wird die Datei `os_memory.c` erweitert.

Beginnen Sie damit, Ihr Konzept der Zustände in der Allokationstabelle zu überarbeiten. Die Tabelle soll für diesen Versuch nicht vergrößert werden. Das bedeutet, dass Sie die neuen Zustände neben den Zuständen aus Versuch 3 (*Heap / Schedulingstrategien*) mit insgesamt vier Bit darstellen müssen. Bedenken Sie die neuen Anforderungen, die den Speicherbereich nicht nur als gemeinsamen Speicher deklarieren müssen, sondern auch bestehende Lese- oder Schreibzugriffe kennzeichnen. Machen Sie sich dazu noch einmal die folgenden Zugriffsreihenfolgen klar, die Abhängigkeiten erzeugen können:

- read-before-write
- write-before-read
- write-before-write

Ihre Implementierung von SPOS soll das gleichzeitige Lesen von mindestens zwei Prozessen erlauben (optional auch mehr). Denken Sie daran, dass der Speicherbereich immer noch aufgrund anderer Lesezugriffe gesperrt sein könnte, wenn der aktuelle Prozess seinen Lesevorgang beendet hat. Es muss also festgehalten werden, wie viele parallele Lesezugriffe zur Laufzeit auf einen Speicherbereich erfolgen. Dies muss ebenfalls mit den vier Bit der Allokationstabelle abgedeckt werden.

Berücksichtigen Sie die nachfolgenden Anforderungen und die Schnittstellenbeschreibung in der zur Verfügung gestellten Doxygen-Dokumentation.

Hinweis

Sie müssen beachten, dass in der Implementierung des Lehrstuhls mehr Funktionen benutzt wurden, als hier explizit gefordert werden. Diese Funktionen sind in der Doxygen-Dokumentation, aber nicht in den Versuchsunterlagen aufgeführt. Sie müssen sie nicht in dieser Form übernehmen. Es empfiehlt sich jedoch, da sie den Umgang mit den Speicherbereichen vereinfachen.

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` analog zu `os_malloc` und `os_free`.
Die Funktion `os_sh_malloc` soll mit dem entsprechenden Treiber und der Anzahl zu allozierender Bytes als Parameter aufgerufen werden und einen Zeiger auf den allozierten Speicherbereich zurückliefern.

Die Funktion `os_sh_free` soll (neben dem Treiber für den SRAM) einen Zeiger auf einen `MemAddr`-Datentyp als Parameter übergeben bekommen und den dazugehörigen Speicherbereich wieder freigeben.

Bei gemeinsamen Speicherbereichen muss es **nicht** möglich sein, zurückverfolgen zu können, welche Prozesse einen Speicherbereich alloziert haben. Es genügt hingegen zu wissen, dass es ein Speicherbereich ist, der von mehreren Prozessen genutzt wird.

Achtung

Die Funktion `os_sh_free` erhält, wie einige andere Funktionen zur Verwaltung des gemeinsamen Speichers, einen Zeiger auf `MemAddr` als Argument. Sie hat also eine leicht geänderte Funktionssignatur im Vergleich zur `os_free`-Funktion. Eine nähere Erläuterung finden Sie im Abschnitt 4.4.1.

Speicherblockierung

Um den Zugriff auf gemeinsame Speicherbereiche zu koordinieren, dürfen Prozesse nicht direkt auf diese zugreifen, sondern sollen Methoden des Betriebssystems aufrufen, die das Lesen und Schreiben für diese sicher realisieren. Da diese Methoden die zuvor genannten Abhängigkeiten berücksichtigen müssen, werden zunächst die Funktionen für die Lese- und Schreibsperren realisiert: Dadurch wird verhindert, dass ein Prozess in einen Speicherbereich schreibt, während ein anderer Prozess auf diesen zugreift.
Implementieren Sie hierfür die folgenden Funktionen:

- `os_sh_readOpen`
- `os_sh_writeOpen`
- `os_sh_close`

Ein Aufruf von `os_sh_readOpen` oder `os_sh_writeOpen` sperrt den Speicherbereich und ein Aufruf von `os_sh_close` gibt diesen Speicherbereich wieder frei. Bedenken Sie, dass, wenn der Speicherbereich von `os_sh_writeOpen` geöffnet wurde, es für einen weiteren Prozess nicht möglich sein darf, denselben Speicherbereich zu öffnen, egal für welche Art von Zugriff. Im Gegenzug sollte es jedoch möglich sein, dass mehr als ein Lesezugriff zur gleichen Zeit auf einem Speicherbereich durchgeführt werden kann.

Hinweis

Die hier genannten Funktionen könnten sinnvollerweise privat deklariert werden, da sie nur von anderen Funktionen innerhalb der `os_memory.c` aufgerufen werden sollten, jedoch nicht direkt aus den Anwendungsprogrammen. Um die Überprüfung Ihrer Methoden während des Versuches zu ermöglichen, sollen Sie dies **nicht** tun, sondern diese Funktionen so implementieren, dass sie global verfügbar sind und somit beliebig durch Prozesse aufrufbar sind.

Lese- und Schreibzugriffe

Der Zugriff auf den gemeinsamen Speicher wird durch die Funktionen `os_sh_read` und `os_sh_write` realisiert. Diese sollen intern die zuvor erstellten privaten Funktionen nutzen, um sicherzustellen, dass alle Lese- und Schreibabhängigkeiten korrekt behandelt werden.

Die Funktion `os_sh_read` soll `length` Bytes nach `dataDest` kopieren. Als Argumente erhält `os_sh_read` einen Treiber (`MemDriver`) und einen Zeiger auf einen `MemAddr`-Datentyp (Statt einer Variablen vom Typ `MemAddr`; Beachten Sie dazu die Achtung-Box auf Seite 9). Soll nicht vom Beginn des angegebenen Speicherbereichs an kopiert werden, kann zusätzlich ein Offset > 0 angegeben werden. Dabei muss überprüft werden, ob ein Lesezugriff überhaupt zulässig ist. Überlegen Sie, welche Größen Sie dazu überprüfen müssen. Für die Funktion `os_sh_write` sind ebenfalls entsprechende Überlegungen anzustellen.

In Listing 4.1 finden Sie ein Beispiel, welches den Zugriff auf den gemeinsamen dynamischen Speicher zeigt. Ein Prozess, der die Befehle aus diesem Listing ausführt, legt zunächst den gemeinsamen Speicherbereich `ptr` mit der Länge sechs Byte und den lokalen Speicherbereich `str` mit der Länge drei Byte an. Die drei Bytes von `str` werden mit den `char`-Werten `'g'`, `'h'` und `'i'` initialisiert. Dann wird die Zeichenkette `str2` mit dem Inhalt `"xij"` lokal angelegt. Die Positionierung dieser Zeichenkette im Arbeitsspeicher wird durch den Compiler bestimmt. Als letztes wird der private dynamische Speicherbereich `control` der Länge vier Byte angelegt. Nach dem Anlegen der Variablen werden zwei Schreibzugriffe und ein Lesezugriff auf den gemeinsamen dynamischen Speicher durchgeführt, die Ihnen die Syntax zur Verwendung der Zugriffsfunktionen demonstrieren sollen. Nach einer nachträglichen Nullterminierung der in `control` gespeicherten Zeichenkette wird diese ausgegeben.

```
1  // 6 Byte gemeinsamen dynamischen Speicher anfordern
2  MemAddr ptr = os_sh_malloc(intSRAM, 6);
3  // 3 Byte lokaler Speicher
4  unsigned char str[3];
5  str[0] = 'g'; str[1] = 'h'; str[2] = 'i';
6  // 4 Byte Speicher auf dem statischen Heap,
7  //   der durch den Compiler reserviert wird
8  unsigned char* str2 = (unsigned char*)"xij";
9  // 4 Byte privaten dynamischen Speicher anfordern
10 MemAddr control = os_malloc(intSRAM, 4);
11 // 3 Zeichen an den Anfang von ptr schreiben
12 os_sh_write(intSRAM, &ptr, 0, str, 3);
13 // 3 Zeichen ab der vierten Stelle von ptr schreiben
14 os_sh_write(intSRAM, &ptr, 3, str2, 3);
15 // 3 Zeichen ab der dritten Stelle von ptr lesen
16 os_sh_read(intSRAM, &ptr, 2, control, 3);
17 // Nullterminierung von Zeichenketten beachten
18 control[3] = 0;
19 // den Inhalt der Variablen control ausgeben.
20 lcd_writeString(control);
```

Listing 4.1: Zugriffsbeispiel auf den gemeinsamen dynamischen Speicher

Lernerfolgsfragen

- Was wird im Zugriffsbeispiel in Listing 4.1 in den gemeinsamen Speicher geschrieben?
- Welche Zeichenfolge wird in der letzten Zeile des Zugriffsbeispiels in Listing 4.1 auf dem Display ausgegeben?
- Auf welche Ereignisse muss der Controller bei `os_sh_free`, `os_sh_readOpen` und `os_sh_writeOpen` warten?
- Was sind die Unterschiede der im Zugriffsbeispiel verwendeten Arten, private Daten abzulegen? An welcher Stelle im SRAM erwarten Sie diese Daten?
- Welche Randbedingungen müssen die Funktionen `os_sh_read` bzw. `os_sh_write` beachten?

Referenzieren gemeinsamen Speichers

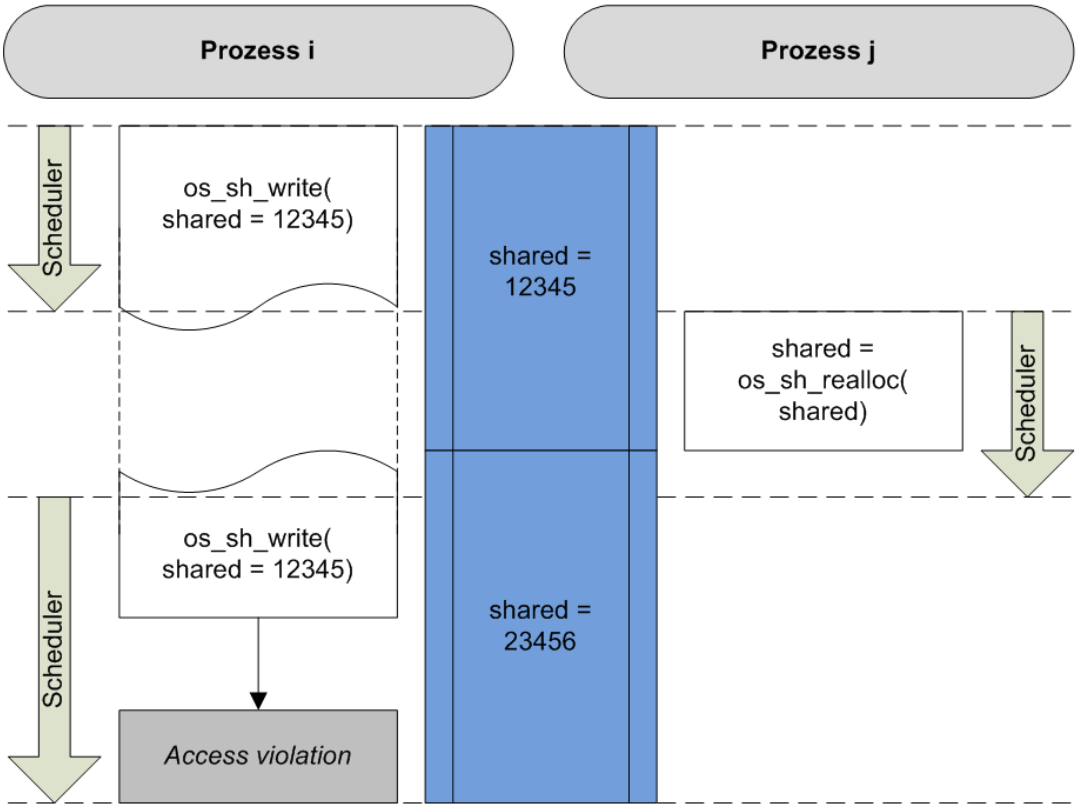
Im folgenden Abschnitt wird erläutert, warum es an einigen Stellen notwendig ist, einen Zeiger auf `MemAddr` an Stelle einer einfachen `MemAddr`-Variablen als Parameter beim Funktionsaufruf zu verwenden.

Hinweis

Variablen vom Typ `MemAddr` stellen eine Adresse im Speicher dar.
Variablen vom Typ `MemAddr*` stellen ein Zeiger auf eine Adresse dar, jedoch keinen direkten Zeiger auf den Speicherbereich.

Im Gegensatz zur bisher implementierten Speicherverwaltung wird in diesem Versuch einigen Methoden ein Zeiger auf eine `MemAddr`-Variable anstatt der Variable selbst übergeben. Dies ist nötig, da durch das präemptive Scheduling ansonsten undefinierte Zustände entstehen können.

Machen Sie sich an folgendem Beispiel klar, weshalb Zeiger benutzt werden müssen: Zwei Prozesse i und j teilen sich einen gemeinsamen Speicherbereich. Die Adresse dieses Speicherbereichs ist in der Variable `shared` vom Typ `MemAddr` gespeichert. Prozess i benutzt eine Zugriffsfunktion auf den gemeinsamen Speicherbereich unter direkter Verwendung der Variable `shared`. Während des Zugriffs, jedoch noch bevor der Speicherbereich gesperrt wurde, entzieht der Scheduler Prozess i die Kontrolle und gibt diese an Prozess j weiter. Prozess j stellt fest, dass der Speicherbereich nur unzureichend groß ist und alloziert einen neuen, größeren Bereich. Prozess j überschreibt die Adresse in der Variablen `shared` mit der Adresse des neuen (größeren) gemeinsamen Speicherbereichs. Wenn die Kontrolle wieder an Prozess i übergeben wird, arbeitet dieser den Aufruf der Zugriffsfunktion ab. Die Änderung der Adresse ist Prozess i jedoch nicht bekannt, da die veraltete Speicheradresse aus der Variable `shared` für die Zugriffsfunktion kopiert[1] wurde und die Zugriffsfunktion nur mit ihrer (inzwischen veralteten) Kopie arbeitet. Prozess i wird also versuchen, auf einen leeren oder anderweitig neu vergebenen Speicherbereich zuzugreifen.



[1] Dieses Verhalten wird *Call-By-Value* genannt.

Wird an Stelle der Variable selbst ein Zeiger auf `shared` (also `&shared`; Typ `MemAddr*`)[2] übergeben, so kennen beide Prozesse immer die aktuelle Adresse des gemeinsamen Speicherbereichs, da sich der Zeiger auf die Variable `shared` nicht ändert.

Um dieses Problem zu vermeiden, erhalten die Zugriffsfunktionen für gemeinsamen Speicher den Wert `&shared` als Argument und müssen diesen bei jedem einzelnen Zugriff neu auswerten. Wird `&shared` nur zu Beginn der Funktion einmal dereferenziert, um den Wert der Variable `shared` auszulesen und lokal abzuspeichern, tritt das oben beschriebene Fehlverhalten weiterhin auf.

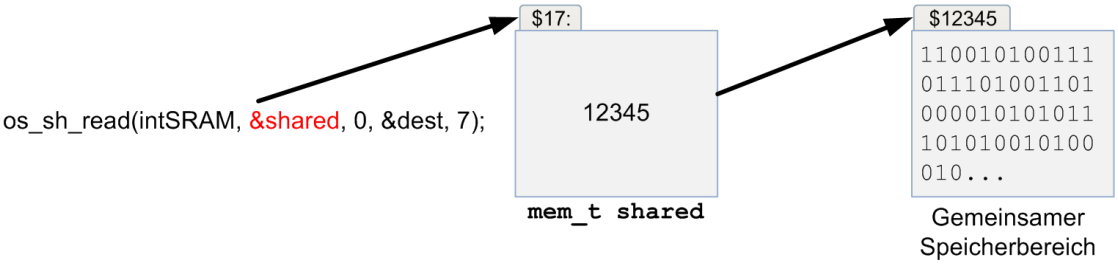


Abbildung 4.1: Veranschaulichung der Nutzung eines Zeigers auf die Variable `shared`

Abbildung 4.1 veranschaulicht die korrekte Nutzung der Funktion `os_sh_read`. Beim Aufruf der Funktion wird die Adresse der Variable `shared` mit dem Wert `$17` übergeben. Die Funktion `os_sh_read` liest bei jedem Zugriff die Adresse des gemeinsamen Speicherbereiches (`$12345`) aus der Variablen `shared` aus. Selbst wenn sich der Inhalt von `shared` ändern sollte, kennt die Funktion `os_sh_read` bei jeder Verwendung der Variable `&shared` die aktuelle Adresse des gemeinsamen Speicherbereichs.

Achtung

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

[2] Dieses Vorgehen entspricht dem Verhalten *Call-By-Reference*.

Übersicht der Signaturen der zu implementierenden Funktionen

Folgende Übersicht soll Ihnen helfen, bei der Implementierung aller zu realisierenden Funktionen, die der Erweiterung des Speichermanagements dienen, die korrekten Signaturen zu verwenden:

- Allokation und Freigabe:
  - `MemAddr os_sh_malloc(MemDriver *driver, uint16_t size);`
  - `void os_sh_free(MemDriver const *driver, MemAddr *ptr);`
- Öffnen und Schließen:
  - `void os_sh_readOpen(MemDriver const *driver, MemAddr const *ptr);`
  - `void os_sh_writeOpen(MemDriver const *driver, MemAddr const *ptr);`
  - `void os_sh_close(MemDriver const *driver, MemAddr const *ptr);`
- Lesen und Schreiben:
  - ```
    void os_sh_write(MemDriver const *driver,
                     MemAddr const *ptr,
                     uint16_t offset,
                     MemValue const *dataSrc,
                     uint16_t length);
    ```
  - ```
    void os_sh_read(MemDriver const *driver,
                    MemAddr const *ptr,
                    uint16_t offset,
                    MemValue const *dataDest,
                    uint16_t length);
    ```

4.4.2 Allokationsstrategien

In diesem Versuch werden Sie alle in Abschnitt 4.3.3 vorgestellten Methoden zur Ermittelung eines passenden Speicherbereiches implementieren, falls Sie diese nicht schon im vorherigen Versuch implementiert haben. Dabei soll es möglich sein, für jeden Speicher eine eigene unabhängige Allokationsstrategie auszuwählen. Berücksichtigen Sie dies bei der Implementierung.

4.4.3 Vorzeitige Abgabe der Rechenzeit eines Prozesses

Implementieren Sie die Funktion `void os_yield(void)`, mit der ein Prozess die übrige Rechenzeit abgeben kann.

Die Funktion `os_yield` ruft den Scheduler so schnell wie möglich auf. Befassen Sie sich dazu noch einmal mit Interruptserviceroutinen und deren Aufruf. Das Datenblatt des ATmega 644 liefert Ihnen in Kapitel *10. Interrupts* alle nötigen Informationen.
Bedenken Sie, dass in diesem Fall derselbe Prozess nicht erneut vom Scheduler ausgewählt werden darf. Es bietet sich an, zu diesem Zweck dem Prozess, der die Rechenzeit

abgibt, den Status `OS_PS_BLOCKED` zuzuweisen. Der Scheduler muss Prozesse mit diesem Status ignorieren. Der Status `OS_PS_BLOCKED` muss nach dem Scheduling wieder zurückgesetzt werden, da blockierte Prozesse nur ein Mal vom Scheduling ausgeschlossen werden.

Implementieren Sie `os_yield` als kritischen Bereich. Stellen Sie vor dem manuellen Aufruf des Schedulers folgendes sicher:

- Er muss auch weiterhin automatisch aufgerufen werden
- Sie haben den aktuellen Zustand der kritischen Bereiche lokal gespeichert, d.h.:
  - Anzahl an geöffneter kritischer Sektionen
  - Zustand des `SREG` Registers

Sobald wieder in `os_yield` zurückgekehrt wird, muss der vorherige Kontext wiederhergestellt werden. Denken Sie in diesem Rahmen an das `OCIE2A` Bit des `TIMSK2` Registers, das von den Funktionen zur Verwaltung kritischer Bereiche verwendet wird.

Achtung

Die Manipulation von Prozess-Zuständen birgt die Gefahr von Seiteneffekten. Blockierte Prozesse dürfen erst nach der Auswahl des nächsten Prozesses wieder auf `OS_PS_READY` gesetzt werden.

Hinweis

Es ist sinnvoll, die `os_yield` Methode nicht nur bei Schreib-/Leseblockaden, sondern auch an anderen Stellen im Betriebssystem zu verwenden. Die Funktion kann z.B. die Warteschleife des Taskwrappers ersetzen. In diesem Fall ist es ein Fehler, den Prozess als blockiert zu kennzeichnen, da er gerade terminiert ist.

4.4.4 Schedulingstrategie

Sie haben in einem vorherigen Versuch die Strategie *InactiveAging* implementiert. Überprüfen Sie noch einmal genau, ob Ihre Implementierung korrekt ist, da diese Strategie eine **wichtige** Grundlage für zukünftige Versuche darstellt. Gleichen Sie Ihre Implementierung im Zweifel mit den entsprechenden Unterlagen ab.

4.5 Präsenzaufgaben

Sie erhalten während des Versuches von den Betreuern eine Sammlung von Testprogrammen, mit denen die Funktionalität Ihres Betriebssystems zum aktuellen Entwicklungsstand getestet wird. Wenn Ihre Anwendungsprogramme fehlerfrei unterstützt werden, testen Sie diese Referenzprogramme. Wenn es mit diesen Programmen Probleme gibt, haben Sie wahrscheinlich bestimmte Anforderungen nicht erfüllt oder gewisse Sonderfälle nicht abgefangen. Ergänzen Sie Ihr Projekt dann entsprechend.

Die Implementierungshinweise, Achtung-Boxen und Lernerfolgsfragen in diesen Unterlagen weisen meist auf notwendige Kriterien für den erfolgreichen Durchlauf der Testprogramme hin.

4.6 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

Pinbelegung für Versuch 4.