Praktikum Technische Informatik

T3-4

Betriebssystem-Konzepte

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
D22527 Hamburg

1 Betriebssystem-Konzepte

Inhalt dieses Versuchs sind die Grundkonzepte von Betriebssystemen. Leider lassen sich wesentliche Eigenschaften von Betriebsssystemen wie die Prozessverwaltung mit Scheduling, virtueller Speicher, Gerätetreiber, Dateisysteme etc. wegen ihrer Komplexität kaum in Praktikumsversuchen studieren. Zudem ist ein Herumspielen mit diesen Komponenten „am lebenden Objekt“ in einem aktuellen Betriebssystem wie Windows oder Unix/Linux nur bedingt zu empfehlen, da schon kleinste Fehler schwere Folgen nach sich ziehen.

Daher bleibt wieder nur die Simulation, um wenigstens einige ausgewählte Themen an stark vereinfachten Beispielen untersuchen zu können. Da die Organisation und das Verhalten der Dateisysteme aus dem täglichen Gebrauch bekannt sein dürfte, behandeln die Versuche die folgenden Themen: die Reaktion auf externe Ereignisse mit Interrupts, die Speicherverwaltung und virtuellen Speicher, und schließlich das Grundkonzept von Prozessen und Prozeßumschaltung. Für eine vertiefte Darstellung sei noch einmal auf die Literatur (z.B. Tanenbaum, Moderne Betriebssysteme) hingewiesen.

2 Interrupt-Verarbeitung

Zentrale Bedeutung für die Verwaltung von I/O-Geräten und die Unterstützung von Multitasking hat das *Interrupt*-Konzept. Die Grundidee dabei ist es, als Reaktion auf ein externes Ereignis das gerade laufende Programm zu unterbrechen und ein neues Programm (die Interrupt-Routine) auszuführen. Durch geeignete Hardware- und Softwareunterstützung kann das unterbrochene Programm später fortgesetzt werden.

Als Demonstration der wesentlichen Merkmale wollen wir jetzt Interrupts für den D·CORE-Prozessor realisieren. Im Steuerwerk sind dabei verschiedene Erweiterung notwendig, um Interrupts verarbeiten zu können. Die einfachste Variante ist im Hades-Design `processor-irq-io.hds` realisiert:

- Der erste Schritt der Befehl-Holen-Phase wird modifiziert. Sobald eine Interrupt-Anforderung detektiert wird, wird der aktuelle Wert des `PC` im `EPC`-Register gespeichert und der nächste Befehl von der (festen) Adresse der Interrupt-Routine geholt.
- Damit auch Impulse auf dem Interrupt-Signal erkannt werden können, wird die Interrupt-Leitung über ein SR-Flipflop `IRQ` geführt, das auch einen kurzen Impuls dauerhaft speichert. Erst innerhalb der Interrupt-Behandlung wird das Flipflop zurückgesetzt.
- Damit bei konstant anliegendem Interrupt-Signal nicht jedes Befehl-Holen wieder von neuem die Interrupt-Routine anspringt, setzt der Prozessor das `IRQE` (interrupt enable) Flipflop zurück, sobald die Interrupt-Aufforderung bearbeitet wird. Das `IRQE`-Flipflop hat Vorrang vor dem `IRQ`-Flipflop.
- Außerdem ändert sich die Initialisierung des Rechners nach einem Reset, da zusätzlich die `IRQ`- und `IRQE`-Flipflops in einen definierten Zustand gebracht werden müssen. Dazu wird in den ersten beiden Takten nach dem Reset (ab Adresse `0`) zunächst `IRQE` gesetzt und anschliessend sofort zurückgesetzt. Die Mikroprogrammschritte für die Fetch-Phase beginnen daher erst ab Adresse `0x02`).
- Es wird ein neuer Befehl `RFI` (return from interrupt) für den Rücksprung aus der Interrupt-Routine benötigt.

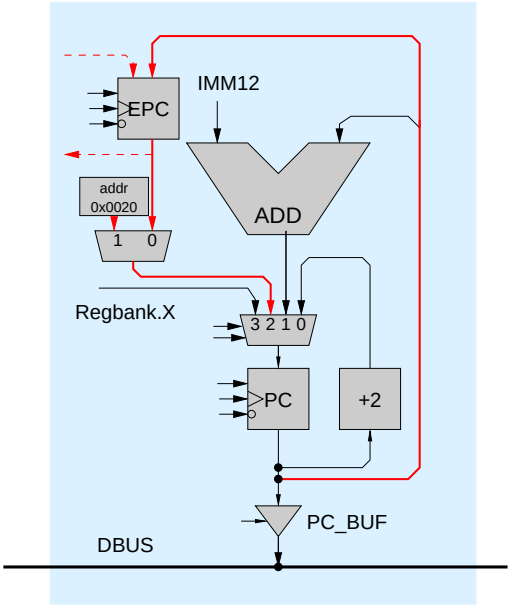

Abbildung 1: Zusätzliche Hardwarekomponenten für die Interruptverarbeitung. Das Register `EPC` speichert bei einem Interrupt den Wert des `PC`, so daß das unterbrochene Programm später fortgesetzt werden kann. Die Interrupt-Startadresse ist konstant (hier `0x0020`).

Die im Zusammenhang mit dem Programmzähler zusätzlich notwendigen Hardwarekomponenten sind in Abbildung 1 dargestellt (rot bzw. fett hervorgehoben). Das Register `EPC` dient dazu, den Wert des `PC` zu Beginn der Interruptbehandlung zu sichern. Gleichzeitig wird die Startadresse des Interrupt-Handlers über den 2:1-Multiplexer in den `PC`geladen. Die anschliessende Fetch-Phase holt dann den nächsten Befehl von genau dieser Adresse. Über den 2:1-Multiplexer kann der Wert aus dem `EPC` wieder zurück in den `PC` geladen werden, um das ursprüngliche Programm an der richtigen Stelle fortzusetzen.

Mit diesen kleinen Hardwareänderungen können bereits Interrupts bearbeitet werden. Der zugehörige erweiterte Microcode muss dabei noch von der Datei `microcode-irq.rom` in das Mikroprogramm-ROM geladen werden.

Natürlich sind diese Funktionen auch im Emulator integriert, so dass Sie diese Aufgaben auch alle mit dem Emulator bearbeiten können.

Aufgabe 2.1: Kennenlernen Erweitern Sie Ihr Programm zur Speicherinitialisierung aus Aufgabe T3-2-2.4 (weil dieses schön lange läuft) um eine zusätzliche Interrupt-Routine, die ein bis dahin unbenutztes Register inkrementiert. Starten Sie das Programm und lösen Sie dann über den IRQ-Schalter Interrupts aus.

Label	Adresse	Befehlscode	Mnemonic	Kommentar
interrupt:	0020			
	0022			
	0024			
	0026			

Aufgabe 2.2: Register sichern Die Interrupt-Routine muss natürlich davon ausgehen, dass alle Register des Prozessors bereits vom Anwendungsprogramm benutzt werden. Es ist daher erforderlich, alle von der Interrupt-Routine benötigten Register zunächst auf den Stack zu sichern und vor dem `RFI` wieder herzustellen:

```
irq:                    ; Start des Interrupt-Handlers
        save_registers  ; Register auf den Stack sichern
        ...             ; hier die Befehle zur
        ...             ; Interrupt-Behandlung
        restore_registers ; Register wiederherstellen
        rfi             ; return from interrupt
```

Da die D·CORE-Hardware kein Register fest für den Stackpointer reserviert, ist an dieser Stelle eine Konvention notwendig, welches Register als Stackpointer reserviert wird (in den bisherigen Programmen war dies immer `R0`). Machen Sie sich klar, dass eine einmal gewählte Konvention von *allen* Programmen und Funktionen auf dem jeweiligen Rechner und Betriebssystem eingehalten werden muss! Schreiben Sie den entsprechenden Code, der alle Register auf den Stack sichert und wiederherstellt.

Aufgabe 2.3: Uhr Fügen Sie einen neuen Taktgenerator (`popup->create->io->ClockGen`) in das Design ein, und schliessen Sie diesen an die Interrupt-Leitung des Prozessors an. Stellen Sie den Taktgenerator auf 1 Hz, so dass der Prozessor einmal pro Sekunde einen Interrupt bekommt. Schreiben Sie dann einen Interrupt-Handler, der eine Uhr in Software nachbildet und die Uhrzeit dann auf das Terminal ausgibt. Als Hauptprogramm können Sie eine einfache Endlosschleife verwenden. (Ein `halt`-Befehl ist in diesem Fall nicht günstig, da der Prozessor daraus auch durch einen Interrupt nicht wieder aufwacht):

```
reset:                  ; Resetadresse
        br start        ; weiter bei Start

interrupt:
        save_registers  ; Register sichern
        ...             ; Uhr: Sek/Min/Std. inkrementieren
        ...             ; Wert aufs Terminal schreiben
        restore_registers; Register wiederherstellen
        rfi             ; Rücksprung

start:  ...             ; Uhr initialisieren, z.B.
        ...             ; R0: Stackpointer
        ...             ; R3: Stunden, R4: Minuten, R5: Sekunden

idle:                   ; Hauptprogramm: nach der Initialisierung
        br idle         ; ist nichts mehr zu tun
```

3 Prozesse und Multitasking

Eine zentrale Eigenschaft aller modernen Betriebssysteme ist das gleichzeitige Ausführen mehrerer *Prozesse* bzw. Programme.

Aufgabe 3.1: Prozesse Zählen Sie auf, durch welche Register und Datenstrukturen ein Prozess gekennzeichnet wird. Welche dieser Werte müssen also bei einem Prozesswechsel vom Betriebssystem mindestens gespeichert werden?

Notwendige Daten für einen Prozeß:

Aufgabe 3.2: Kooperatives Multitasking Beim *kooperativen Multitasking* ist jeder einzelne Prozess am Scheduling beteiligt. Die Grundidee ist, dass ein laufender Prozess freiwillig (zum Beispiel nach Abarbeiten einer Programmschleife oder Funktion) die Kontrolle an das Betriebssystem zurückgibt. Dieses schaltet dann auf den nächsten Prozess um. Welche Vor- und Nachteile hat dieses Verfahren?

Vorteile:

Nachteile:

Aufgabe 3.3: Präemptives Multitasking Um die Systemstabilität zu erhöhen, arbeiten die meisten modernen Betriebssysteme mit dem sogenannten *präemptiven* Multitasking, bei dem das Betriebssystem die Zeitscheiben für die einzelnen aktiven Prozesse zuteilt. Die notwendige Unterbrechung des jeweils aktiven Prozesses wird dabei durch regelmässige Timer-Interrupts vorgenommen.

Beim Ausführen eines Interrupts wird der Wert des `PC` automatisch in das Register `EPC` übertragen. Vor dem Umschalten auf einen neuen Prozeß muss dieser Wert natürlich gerettet werden, damit der unterbrochene Prozeß später fortgesetzt werden kann. Dies ist mit der bisher verwendeten Hardware aber gar nicht möglich (warum?).

Wir führen deshalb einen neuen, zusätzlichen Befehl `EEPC` ein (*exchange EPC and R2*), der den Inhalt der Register `EPC` und (willkürlich gewählt) `R2` austauscht. Zur Befehlskodierung wird dabei der bisher nicht verwendete Opcode `6***` benutzt. Möglich wird dies in der Hardware durch die zusätzlichen, in Abb. 1 gestrichelt eingezeichneten Datenpfade.

Aufgabe 3.4: Multitasking-Demo: Zwei Zählprogramme Um das Umschalten überhaupt demonstrieren zu können, sind natürlich mehrere „Nutz“-Prozesse notwendig. Um die Aufgabe möglichst einfach zu halten, schreiben Sie am besten zwei Kopien des `count`-Programms, die jeweils auf eine eigene Variable im RAM zugreifen. Das folgende Codebeispiel geht davon aus, dass der erste Prozeß den Inhalt der Speicherstelle `0x8100` inkrementiert und den aktuellen Wert auf die LEDs ausgibt, während der zweite Prozeß den Inhalt von `0x8102` inkrementiert und den aktuellen Wert auf das Terminal schreibt. Mit dieser Wahl können Sie dann auch direkt sehen, welcher Prozeß gerade aktiv ist.

Der Ablauf des gesamten Programms ist also:

```
init:     ...               ; Initialisierung des "Betriebssystems"
          ...               ; soweit notwendig
          br start


interrupt:                  ; INT-Handler: hier Prozesswechsel
          save_registers    ; Register (Prozess i) sichern
          ...
          ...               ; hier der Prozesswechsel:
          ...               ; PC / Stackpointer sichern
          ...
          restore_registers ; Register (Prozess i+1) holen
          rfi               ; Rücksprung, d.h. Prozess i+1
                            ; fortsetzen

start:    ...               ; Laden der einzelnen Programme
          ...               ; hier: Zähler 1/2 / Idle / INT-Handler
          ...               ; u.a. Startadresse, Stackadresse, ...
          br idle


idle:                        ; "Idle"-Prozess: nichts tun
          br idle            ; Endlosschleife


count1:   ...                ; "Count"-Prozess1:
          ...                ; Zähler an Adresse 0x8100 inkrementieren
          ...                ; aufs Terminal ausgeben
          br count1          ; Endlosschleife

count2:   ...                ; "Count"-Prozess2:
          ...                ; Zähler an Adresse 0x8102 inkrementieren,
          ...                ; auf die LEDs ausgeben
          br count2          ; Endlosschleife
```

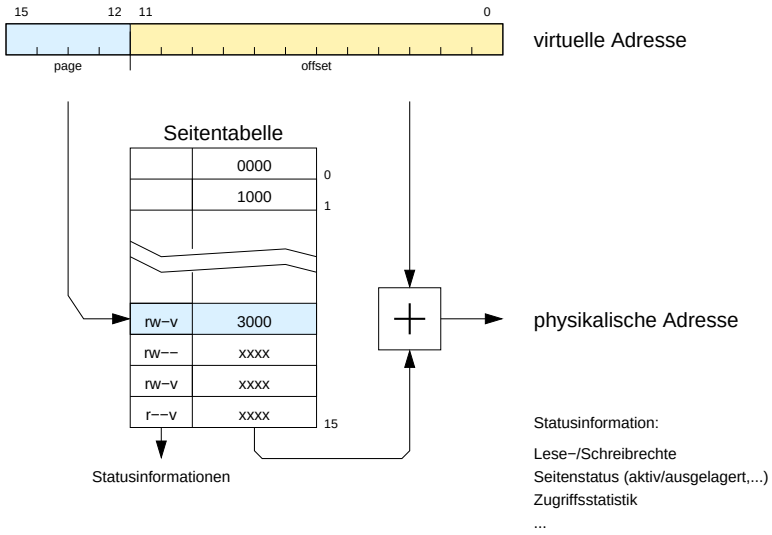

Abbildung 2: Adressumsetzung von virtuellen in physikalische Adressen mit Paging: der obere Teil der Adresse dient als Index in die Seitentabelle, der adressierte Inhalt liefert die oberen Bits der physikalischen Adresse und Statusinformationen.

4 Virtueller Speicher

Alle bisher im Praktikum verwendeten Programme arbeiten direkt mit den im System vorhandenen, *physikalischen* Adressen. Um mit diesem Verfahren ein Multitasking zu erlauben, müssen alle beteiligten Programme/Prozesse aufeinander abgestimmt sein, um Konflikte durch mehrfache Belegung von Speicheradressen zu vermeiden. Zum Beispiel kann ein bestimmtes Programm nicht einfach mehrfach gestartet werden, ohne alle benutzen Adressen abzuändern. Auch müssen bei jeder Änderung des Systems (z.B. der Adressen von I/O-Bausteinen oder einer Speichererweiterung) alle Programme entsprechend angepasst werden.

Diese Nachteile lassen sich mit dem Konzept des *virtuellem Speichers* vermeiden. Dabei darf jeder einzelne Prozeß den gesamten virtuellen Adreßraum ausnutzen. Die Umsetzung der virtuellen Programmadressen auf die physikalischen Adressen (der im System vorhandenen Hardware) erfolgt mit einer Speicherverwaltungseinheit (MMU), die zentral vom Betriebssystem verwaltet wird. Dabei wird fast immer das Konzept des *Paging* benutzt, wobei der Adressraum in Seiten fester Größe unterteilt wird, so daß die Umsetzung nur die Seitennummer, nicht aber die fortlaufenden Adressen innerhalb der Seite betrifft (siehe Abb. 2). Durch das Auslagern von Speicherseiten auf einen Hintergrundspeicher ist es zusätzlich möglich auch Programme auszuführen, deren Code und Daten zu groß sind, um auf einmal in den verfügbaren Hauptspeicher hineinzupassen.

Aufgabe 4.1: Inhalt der Seitentabelle Zählen Sie auf, welche zusätzlichen Daten und Statusinformationen von „richtigen“ Betriebssystemen in der Seitentabelle verwaltet werden:

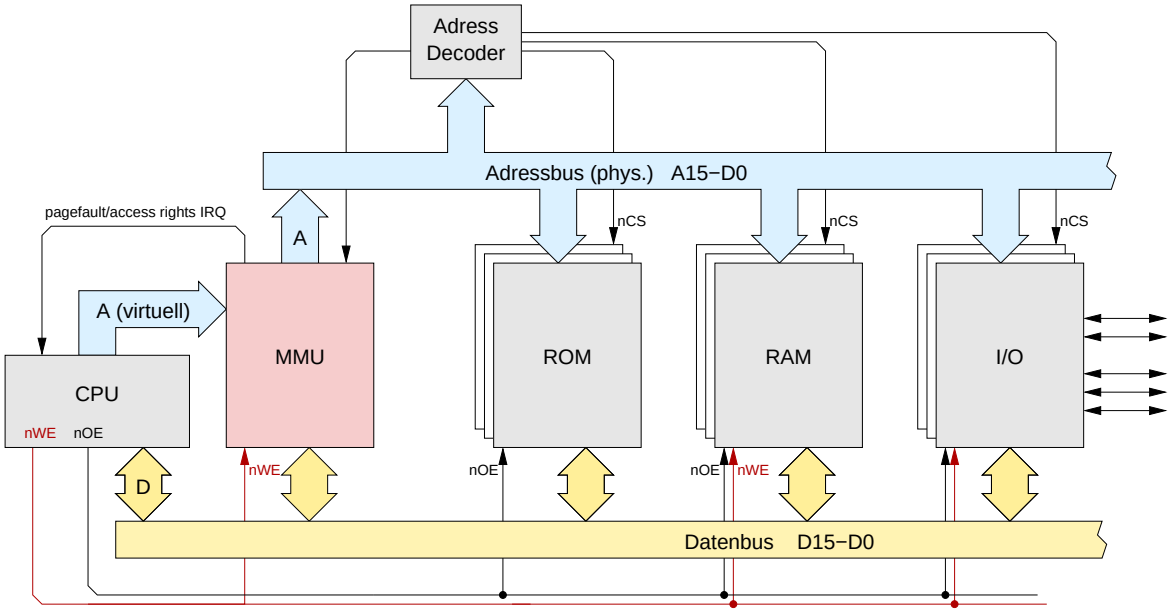

Abbildung 3: Virtueller Speicher durch Erweiterung um eine Memory-Management-Unit, MMU

Aufgabe 4.2: Algorithmen Wie funktionieren die im folgenden genannten Algorithmen zur Seitenauslagerung? Welche sind empfehlenswert, welche sind praktisch realisierbar?

Zufällige Ersetzung (Random):

FIFO:

LFU:

LRU:

Aufgabe 4.3: Größe der Seitentabelle Leider kann man die in Abbildung 2 skizzierte, einstufige Seitentabelle nur bei kleinen Systemen realisieren. Welche Größe hätte eine einstufige Seitentabelle auf einem 64-bit Prozessor mit 64-bit Adressen bei einer Seitengröße von 4 KB?

Größe der Seitentabelle im Beispiel:

Welche zwei Alternativen zur einfstufigen Seitentabelle werden statt dessen verwendet?

Algorithmus 1:

Algorithmus 2:

Aufgabe 4.4: Einfache MMU für D·CORE Für den folgenden Versuch wird das D·CORE-System um eine einfache MMU ergänzt. Um die Speicherverwaltung auf das Minimum zu reduzieren, teilen wir den Hauptspeicher in insgesamt 16 Seiten der Größe 4 KByte auf und verzichten auf alle zusätzlichen Statusinformationen. Zur Umsetzung von virtuellen in physikalische Adressen genügt daher ein kleines RAM mit 16 Worten für die obersten vier Adressbits, während die unteren 12 Bit der Adresse direkt durchgereicht werden. Diese Organisation ist in Abbildung 3 und 4 dargestellt.

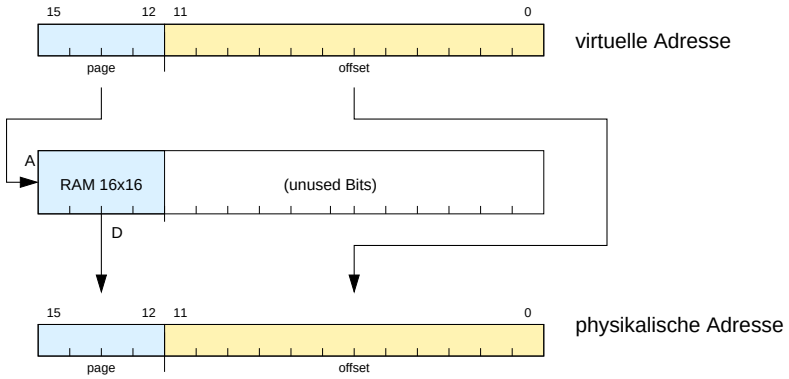

Abbildung 4: Adressumsetzung mit einem RAM der Größe 16x16. Die oberen 4 Bit des RAM-Inhalts dienen als Seitenadresse, die unteren 12 Bits werden ignoriert.

Natürlich muss der Adressdekoder des Systems erweitert werden, damit das RAM für die MMU aktiviert werden kann. Dies erfolgt an den (physikalischen) Adressen `0x7010..0x702E`.

Aufgabe 4.5: Initialisierung der MMU Nehmen Sie an, dass nach dem Einschalten des Systems alle Einträge in der MMU auf den Wert 0 bzw. 7 gesetzt sind (siehe Abbildung 5b). Der Prozessor kann also zunächst nur die Seite 0 zugreifen (wo das ROM liegt) und auf Seite 7 (I/O-Bereich).

Schreiben Sie die Funktion `init_pagetable`, die die MMU „linear“ initialisiert, so daß also jede virtuelle Adresse gleich der physikalischen Adresse ist.

Aufgabe 4.6: Abspeichern der Seitentabelle Schreiben Sie jetzt eine Funktion `store_pagetable`, um den Inhalt der Seitentabelle in den Speicher zu kopieren. Übergeben Sie die Anfangsadresse des gewünschten Speicherbereichs als Pointer in `R10`.

Aufgabe 4.7: Laden der Seitentabelle Schreiben Sie die zugehörige Funktion `load_pagetable`, um die Seitentabelle mit Werten aus dem über `R10` adressierten Speicherbereich neu zu laden.

Aufgabe 4.8: Speicherschutz Da unser System keine expliziten Mechanismen für den Speicherschutz bereitstellt, müssen alle beteiligten Programme kooperieren. Zum Beispiel ist es sinnvoll, dass alle Prozesse die Speicherbereiche der Seiten `0x0000` (Reset und Interrupt-Handler) und `0x7000` (I/O-Bereich, Seitentabelle) gemeinsam nutzen und diese Einträge nicht verändern.

Was passiert, wenn ein Programm in der Seitentabelle den Eintrag für die Adressen `0x7000` verändert?

Jetzt wird es Zeit, alle bisher entwickelten Konzepte zusammen in einem Gesamtsystem zu testen: I/O, Interrupts, virtueller Speicher.

Aufgabe 4.9: Prozeßwechsel Schreiben Sie einen Interrupt-Handler, der bei jedem Aufruf das gerade laufende Programm unterbricht und auf den nächsten Prozess wechselt. Dazu sind offenbar die

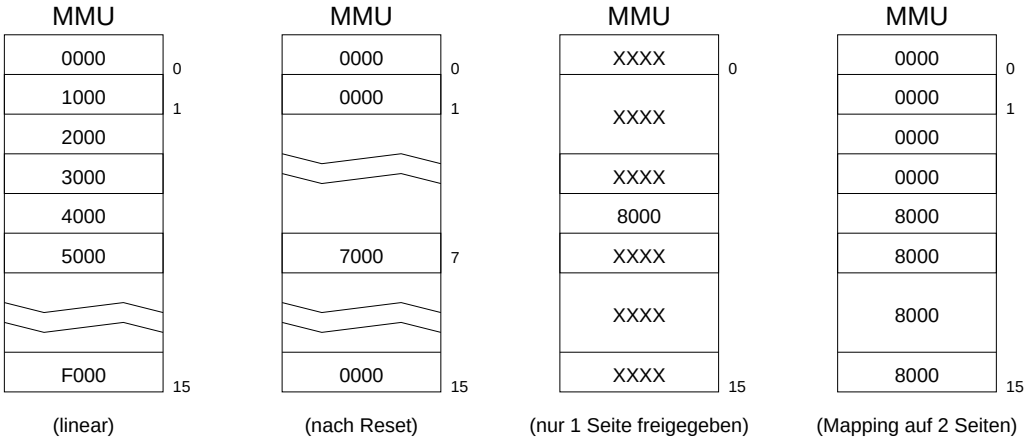

Abbildung 5: Beispiele für die Adressumsetzung: lineares Mapping, Reset-Zustand mit Mapping auf Seite 0 (ROM) und Seite 7 (I/O), Sperren aller Seiten außer einer, Umsetzung aller virt. Adressen auf nur zwei phys. Seiten

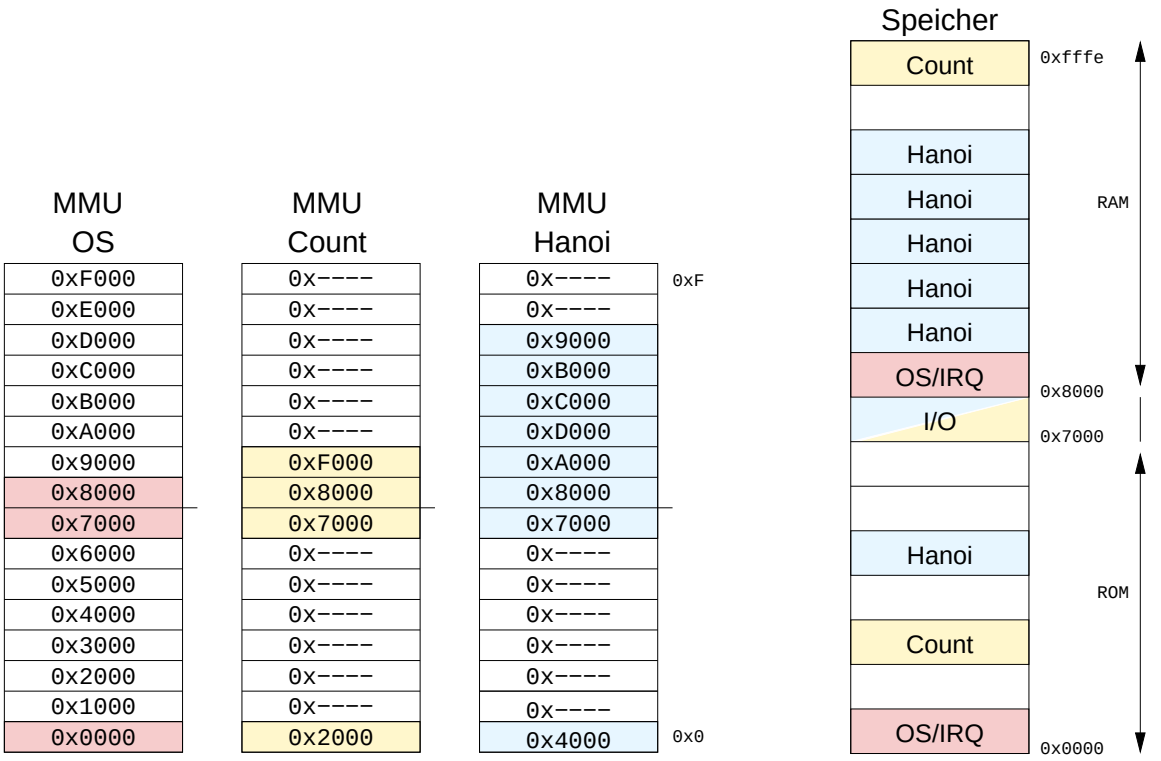

Abbildung 6: Vorschlag für die Speicherorganisation bei gleichzeitiger Verwendung des Türme-von-Hanoi und des Count-Programms. Das Türme-von-Hanoi Programm benötigt viel Speicher für seinen Stack. Das Count-Programm benutzt nur sehr wenig Speicher, muß aber mindestens eine Seite reservieren. Der I/O-Bereich wird gemeinsam benutzt.

folgenden Aktionen notwendig:

1. Sichern der aktuellen Registerinhalte des Prozessors für Prozeß i
2. Abspeichern der aktuellen Seitentabelle für Prozeß i
3. Neuladen der Seitentabelle mit den Werten für den nächsten Prozeß $i+1$
4. Wiederherstellen der Prozessorregister für Prozeß $i+1$
5. Rücksprung vom Interrupt-Handler und damit Fortsetzen von Prozeß $i+1$.

Der wesentliche Unterschied zum Prozeßwechsel in Aufgabe 3 ist das zusätzliche Laden der Seitentabelle. Zum Beispiel kann jetzt ein Programm durchaus mehrfach gestartet werden, wenn die Seitentabelle dafür sorgt, daß die von den einzelnen Instanzen des Programms benötigten Datenbereiche auf unterschiedliche Speicherseiten abgebildet werden. Solange kein selbstmodifizierender Code verwendet wird, muß der Programmcode selbst natürlich nur einmal im Speicher gehalten werden.

Aufgabe 4.10: Datenstrukturen Machen Sie sich klar, welche Datenstrukturen Sie für die Umschaltung benötigen und notieren Sie die von Ihnen gewählten Speicherbereiche hier:

Seitentabelle für den Interrupt-Handler:
Seitentabelle für Prozeß 1:
Seitentabelle für Prozeß 2:

virt. Adressen für den Interrupt-Handler:

Speicherbereich der Register für Prozeß 1:
Speicherbereich der Register für Prozeß 2:

Aufgabe 4.11: Prozessumschaltung Demonstrieren Sie jetzt die in Abbildung 6 skizzierte Speicherverwaltung. Schreiben Sie dazu einen Interrupt-Handler, der bei einem manuell ausgelösten Interrupt zwischen zwei Prozessen umschaltet. Verwenden Sie das Türme-von-Hanoi Programm aus Aufgabe T3-3 und zweitens das einfache Zählprogramm, das seinen Zählerstand auf die LEDs ausgibt.