

Multitasking mit TSS

0. Inhaltsangabe

0. Inhaltsangabe
1. Einleitung
2. Überblick
3. Aufbau eines TSS
4. Task-Tabelle
5. TaskGates
6. Dispatcher
7. Scheduler
8. Scheduling Strategien
9. Taskwechsel
10. Implementierung und Praxis
11. Tipps & Tricks
12. Schluss

1. Einleitung

In diesem Tutorial möchte ich jedem, der gewillt ist, Multitasking in seinem Betriebssystem mit TSS auf Hardwarebasis zu machen, erklären, wie er dies machen kann, bzw. anderen, die nicht beabsichtigen dies zu tun, zeigen, wie Multitasking in der Theorie funktioniert. Bei Fragen oder Problemen bin ich immer unter Email joachim_neu@web.de, ICQ# 247-390-343 oder MSN joachim_neu@web.de erreichbar. Also nicht scheuen und mich bei Ungeklärtem kontaktieren. (Wenn man sich per ICQ meldet, bitte beim Anschreiben nicht „hallo“ sagen, sondern, worum es geht, sonst könnte es sein, dass ich die Nachricht als Spam einschätze.

In diesem Tutorial werde ich zuerst auf die verschiedenen Formen von Multitasking und einen Teil der Theorie, und danach auf TSS-Multitasking speziell eingehen.

2. Überblick

Bei Multitasking gibt es 2 wichtige Teile, die man immer braucht. Der erste Teil ist eine Tabelle, in der alle Tasks aufgelistet sind. Im zweiten Teil wird der aktuelle Taskzustand gespeichert. Es ist wichtig, dass der Computer alle Inhalte der CPU-Register speichert, sodass das Programm vom Taskwechsel nichts merkt und davon nicht beeinflusst wird. Und genau hier setzen die Unterschiede ein. Bei hardwarebasiertem Multitasking speichert die CPU den Registerzustand automatisch in einer anderen Tabelle, dem TSS genannten TaskStateSegment. Beim softwarebasierenden Multitasking wird dieses Speichern vom Betriebssystem vorgenommen und erfolgt meist auf dem programmeigenen Stack. Daraus resultierend entstehen auch Unterschiede der ersten Tabelle. Beim hardwarebasierendem Multitasking ist man nur auf einen Eintrag mit der Deskriptornummer angewiesen. Bei softwarebasierendem Multitasking braucht man zwar dies nicht, allerdings muss man SS und ESP speichern. Meist werden noch ein Speicherbereich für eine eindeutige Task-Nummer, die TaskID und einen Bereich, der den Status des Tasks für das System festhält, benutzt. Je nach Scheduling Strategie benutzt man auch noch einen Speicherbereich für die Priorität des Tasks, sodass man den Task mit der höchsten Priorität häufiger oder länger laufen lassen kann. Selten werden für spezielle Scheduling Strategien auch noch Speicherbereiche zur Sicherung der Startzeit oder ähnlichem benutzt.

Es gibt zwei verschiedene Versionen von Multitasking, kooperatives und preemptives. Kooperatives Multitasking vertraut darauf, dass jedes Programm so lange Zeit bekommt, wie es braucht und danach ein Systeminterrupt auslöst, dass die Rechenleistung an den nächsten Task weiterreicht. Dieses Konzept ist in letzter Zeit hinfällig geworden, da es in Zeiten von Viren und Trojanern nichtmehr das Schlauste wäre, Programmen die zeitunbegrenzte Kontrolle über den PC zu geben, da diese den ganzen Ablauf stoppen könnten. Neue Ideen mussten her, meist in Form von

preemptivem Multitasking. Bei diesem wird dem Programm/Task nach gewissen Zeit vom System durch das TimerIRQ die Kontrolle entzogen. Da dies meiner Meinung nach die einzig sinnvolle Lösung ist, werde ich hier darauf eingehen.

Später werden wir noch einige Teile in Assembler zusammen coden, der dir helfen wird, das ganze zu verstehen.

3. Aufbau eines TSS

Da wir uns jedoch um Multitasking auf Hardwareebene kümmern wollen, werde ich hier den Aufbau eines TaskStateSegments schildern. Den Aufbau werde ich später noch in einer Tabelle verdeutlichen, jedoch werde ich davor noch wichtige Informationen erleutern. Dadurch, dass es äußerst unsicher wäre, den Stack im Userbereich zu belassen, wenn man ein Systeminterrupt oder Systemcall aufruft, da später diese Informationen ausgelesen werden könnten und sowieso Daten im Privileglevel 3 nicht sonderbar geschützt sind, wurden im TaskStateSegment Datenbereiche für Stackdaten eingerichtet. Da der Protected Mode über 4 Privileglevel (0-3) verfügt, Level-3-Daten jedoch sowieso ungeschützt sind, braucht man dafür keinen eigenen Stack, sondern kann den User-Stack benutzen. Es existieren also 3 zusätzliche Stacks, für Privileglevel 0,1 und 2, für die man auch Speicherplatz für die SS-Register und ESP-Register braucht, welcher als SS0, SS1, SS2, ESP0, ESP1 und ESP2 bezeichnet wird. Nun hier aber der Aufbau:

0	0x00	0	BackLink
4	0x04	ESP0	
8	0x08	0	SS0
12	0x0C	ESP1	
16	0x10	0	SS1
20	0x14	ESP2	
24	0x18	0	SS2
28	0x1C	CR3	
32	0x20	EIP	
36	0x24	EFLAGS	
40	0x28	EAX	
44	0x2C	ECX	
48	0x30	EDX	
52	0x34	EBX	
56	0x38	ESP	
60	0x3C	EBP	
64	0x40	ESI	
68	0x44	EDI	
72	0x48	0	ES
76	0x4C	0	CS
80	0x50	0	SS
84	0x54	0	DS
88	0x58	0	FS
92	0x5C	0	GS

0	0x00	0	BackLink
96	0x60	0	LDT
100	0x64	I/O Port Bitmap Base	0 T
104	0x68	Freier Bereich	

So. Nun werde ich den Aufbau genauer Erklären. Die **gelb** hinterlegten Einträge bezeichnen das Offset des Eintrags zum Beginn der Tabelle. Alle **rot** hinterlegten Einträge haben den Wert „0“ und sollten den auch beinhalten. Alle **blau** hinterlegten Einträge bezeichnen vom Programm beeinflussbare Einträge. Alle **lila** hinterlegten Einträge müssen vom Betriebssystem gesetzt werden und sind nicht vom Programm beeinflussbar. Alle **grün** hinterlegten Bereiche werden automatisch gesetzt. Alle **orange** hinterlegten Bereiche stehen frei zur Verfügung. Jetzt werde ich die wahrscheinlich noch unklaren Bereiche erklären. ESPn und SSn hab ich vorher schon erleutert. In BackLink speichert die CPU bei einem Interruptaufruf, bei dem ein TaskGate in der IDT angegeben wurde, die Deskriptornummer des Auslösertasks. Zu diesem kehrt sie bei gesetztem NT-Flag im Flagregister mit einem IRET zurück. Die Bereiche, die die Namen der Register tragen dürften klar sein. Beim Eintrag „I/O Port Bitmap Base“ wird das Offset einer Tabelle angegeben, die angibt, auf welche Portadressen der Task Zugriff hat. Bei dem Wert 0 ist diese deaktiviert, sodass der Zugriff alleine durch das IOPL im Flagregister gesteuert ist. Dann ist da noch das T-Flag. Ist dieses gesetzt, so wird nach jedem Taskwechsel zu diesem Task ein Interrupt 1 ausgelöst. Der Bereich nach der Adresse 0x68 steht dem Systemprogrammierer frei zur Verfügung. Das Limit eines TSS muss also mindestens 0x68 betragen. Nach der eigentlichen TSS-Tabelle ist dann Platz für die I/O-Port-Tabelle oder Werte, die das System vom Task noch braucht und nicht in der Task-Tabelle speichern will. Wie ich schon sagte braucht jeder Task ein TSS und dies muss in der GDT eingetragen werden. Ein TSS hat als die 4 Typ-Bits den Wert „1001“ als Binärzahl. Dies ist der Aufbau eines TSS-Deskriptors:

Limit 0..15											
Basis 0..15											
P	DPL		S	Typ (=1001)			Basis 16..23				
Basis 24..31							G	D	0	AVL	Limit 16..19

Jede Zeile der Tabelle gibt ein Word wieder, eine 16 Bit lange Zahl. Eigentlich gibt es nichts, was unterschiedlich zu dem normalen Aufbau eines Deskriptors wäre, deswegen erkläre ich nur kurz alle Felder. Alle, die mit „Limit“ bezeichnet sind geben die Größe des Segments an. Alle mit „Basis“ bezeichneten geben die Basisadresse im Speicher an. Das Bit „P“ gibt an, ob sich das Segment im Speicher (present) befindet. Es sollte immer auf 1 gesetzt sein, da es meiner Meinung nach nicht zu den schlauesten Lösungen zählt, TSSs auszulagern, die man ständig braucht, zumal diese nicht sehr viel Speicher sparen. Die 2 Bit lange Stelle „DPL“ gibt das DescriptorPrivilegeLevel an. Damit kann man kontrollieren, welche Privilegstufe Programme brauchen, die auf dieses Segment zugreifen wollen. Das Bit „S“ ist das System-Bit. Es unterscheidet spezielle Segmente (wie TSS und alle Gates) von Standardsegmenten (Codesegment, Datensegment). Bei speziellen Segmenten ist es nicht gesetzt, sprich 0. Dann gibt es noch das „G“ (granularity)-Bit, das angibt, ob die Größenangabe des Deskriptors in Bytes (=0) oder in Kbytes (=1) zu verstehen ist. Da ich noch kein TSS mit 1 Kbyte oder mehr Speicherverbrauch gesehen habe, sollte dies den Wert 0 haben. Das „D“-Bit gibt an, ob der Deskriptor zu dem 286er kompatibel ist, oder nicht. Dort empfehle ich den Wert 1, also inkompatibel, dann kann man das Segment vergrößern, ohne sich irgendwann man damit wieder auseinander setzen zu müssen. Das „AVL“-Flag steht zur freien Verfügung. Das „Typ“-Feld haben wir ja schon besprochen, es muss bei einem TSS den Wert 1001b beinhalten.

Wenn ein Task aktiviert wird, so wird sein „Typ“-Feld mit dem Wert „1011“ als Binärzahl belegt. Das zeigt, dass das aktuelle Segment aktiv ist und man nicht zu ihm wechseln kann. Davon wissen wir schon, dass man mindestens zwei Tasks braucht, damit Multitasking mit TSS überhaupt richtig funktioniert. Ebenso wird beim Verlassen eines Tasks der Wert wieder zu 1001b.

4. Task-Tabelle

Desweiteren braucht man noch, wie besprochen, eine Tabelle, die die Deskriptornummern und TaskIDs speichert. Auch der Status wird dort oft gespeichert, da das System ihn braucht, um zum Beispiel pausierte Tasks zu übergehen, und es umständlich wäre, diesen Wert erst aus dem TSS zu lesen, zumal dies nicht als Daten-Segment lesbar ist, was heißt, dass man erst den Platz im Speicher finden und dann über ein anderes Segment auslesen müsste. Außerdem beinhaltet er oft noch eine TaskID, damit man den Task nicht über seine TSS-Nummer identifizieren muss. Da man maximal 8192 TSSs haben kann (maximale Größe der GDT ist 0x10000), allerdings aber noch ein paar Einträge fürs System abgehen, reicht ein Word zum Speichern der TaskID und TSS-Nummer. Als Statusbereich dürfte auch ein Word (16 Bits) reichen. Auch in dieser Tabelle ist eine Zeile ein Word.

TaskID
TSS-Nummer
Status

Es gibt 2 Möglichkeiten, das Ende dieser Tabelle aus Task-Tabellen-Einträgen zu finden.

1. Man speichert in einer Variable, wie viele Einträge man hat und überprüft das aktuelle und das maximale Offset, um einen WrapAround (beim Anfang anfangen) zu vollziehen.
2. Man gibt am Ende der Tabelle dem nächsten leeren Eintrag die ID 0xFFFF oder ein sonstiges „Endezeichen“ und führt einen WrapAround aus, sobald man als ID den Wert des Endezeichens gefunden hat.

Das war eigentlich schon das Wichtigste über die Task-Tabelle.

5. TaskGates

Außerdem gehören zu TSS-Multitasking noch TaskGates. TaskGates kann man sowohl in der GDT als auch in der IDT eintragen. Wenn man sie an erste Stelle stellt, so wird der durch das TaskGate angegebene Task bei der 0. Exception gestartet. Ebenso ist es bei allen anderen Interrupts und/oder Exceptions. Ein TaskGate ist wie folgt aufgebaut: (wieder ist jede Zeile ein Word)

Reserviert (=0)				
TSS-Deskriptor-Nummer				
P	DPL	S	Typ	Reserviert (=0)
Reserviert (=0)				

Alle Bereiche, die „Reserviert“ sind, sollten den Wert „0“ beinhalten. Im Bereich „TSS-Deskriptor-Nummer“ wird der Deskriptor festgehalten, auf den das TaskGate verweist. Die Felder „P“, „DPL“ und „S“ habe ich vorher schon erklärt, sodass dies hier überflüssig ist. Der „Typ“ muss bei einem TaskGate den Wert „0101“ als Binärzahl beinhalten.

TaskGates werden an sich eigentlich mein Multitasking selten verwendet, sondern lediglich als Einträge in der IDT, damit bei einer Exception oder einem Interrupt ein eigener Task aufgerufen wird. Dann wird das NT-Flag gesetzt und die CPU wechselt bei IRET zu dem Task, dessen TSS-Deskriptor im BackLink-Feld eingetragen ist.

6. Dispatcher

So, nun aber nach dem ganzen Aufbauzeug zur Theorie. Ein Taskwechselalgorithmus

sollte idealerweise aus zwei Teilen bestehen, dem Scheduler und dem Dispatcher, um den wir uns nun kümmern wollen. Der Dispatcher ist dafür verantwortlich, dass der Task die CPU zugeteilt und entzogen bekommt. Er muss alle Register laden bzw speichern und den eigentlichen Taskwechsel veranlassen. Beim softwarebasierenden Multitasking besteht diese Aufgabe des Dispatchers darin, zuerst die Werte des aktuellen Tasks auf dessen Stack zu pushen, dann den neuen Stack zu laden und die Werte des Tasks zu popen. Bei hardwarebasierendem Multitasking besteht die Aufgabe lediglich darin, einen Task-Switch zu veranlassen, in dem die CPU dann eigenständig die Register sichert und die neuen lädt.

7. Scheduler

Und hier setzt auch schon der Scheduler an. Während der Dispatcher nur die Taskwerte sichert und die neuen setzt besteht die Aufgabe des Schedulers darin, den nächsten Task auszuwählen. Dies tut er anhand der Scheduling Strategie. Es gibt verschiedene dieser Strategien, welche wir später noch genauer besprechen werden. Bei der einfachsten Strategie sucht der Scheduler einfach nur den nächsten Task in der Task-Tabelle raus und gibt dessen Werte an den Dispatcher weiter. Bei hardwarebasierendem Multitasking ist dies nur die Nummer des TSS-Deskriptors, bei softwarebasierendem Multitasking ist dies SS und ESP des Tasks.

8. Scheduling Strategien

Dies ist einer der umfangreichsten Punkte und der schwierigste, so meinen Experten. Das Problem sei ihrer Meinung nach nicht das Wechseln der Tasks sondern das Auswählen des Tasks, welcher als nächstes zu bearbeiten ist. Ich bin allerdings der Meinung, dass das Problem anfangs eher das Wechseln an sich ist, weshalb man sich nicht mit wahrscheinlichkeitsberechnenden Algorithmen rumschlagen sollte, sondern sich anfangs mit dem einfachsten Verfahren abgeben sollte, einfach den nächsten Task zu nehmen, der in der Tabelle kommt. Allgemein werden an eine Scheduling Strategie folgende Anforderungen gestellt:

1. Sie muss fair sein, was heißt, dass alle Tasks von Grund auf gleich zu behandeln sind, sodass kein Task ständig dran kommt und kein Task nie dran kommt.
2. Sie darf nicht zu komplex sein, sodass sich die CPU mehr mit dieser als mit den Tasks beschäftigen würde.
3. Sie muss unabhängig sein, das heißt, sie darf nicht mit der Anzahl der Tasks zusammenhängen, zum Beispiel nur bei 5 laufenden Tasks funktionieren.
4. Sie sollte berechenbar sein, also nicht auf Zufallswerten basieren oder auf jedem System anders funktionieren.
5. Sie darf nicht beeinflussbar oder manipulierbar sein.

Es gibt viele verschiedene Strategien. Die einen benutzen Prioritätsstufen zum Einteilen der Tasks, andere benutzen die Länge der Tasks. Bei Prioritätseinteilungen gibt es entweder die Möglichkeit, den Tasks die Prioritäten nach ihrem Einsatzbereich (Treiber, Programm,...) zu geben, und dadurch zum Beispiel den Treibern mehr Rechenzeit zuzuteilen, oder aber das Privileglevel zu erhöhen, wenn der Task sich selber pausiert und damit anderen den Vortritt lässt, wenn er gerade keine Rechenzeit benötigt. Sehr häufige Scheduling Strategien sind „Round Robin“, „First Task First“ und „Shortest Time Left“. Diese und einige andere werde ich nun ein bisschen erläutern:

– Round Robin

Bei Round Robin werden alle Tasks in einer Kette hintereinander abgearbeitet und jeder bekommt die gleiche Rechenzeit. Wenn man Prioritäten benutzt, so hat man mehrere Task-Tabellen nach denen man die Tasks aussucht. Die andere Möglichkeit ist, dass es jedes Prioritätslevel nur einmal gibt und ein Task beim Erhöhen seines Levels einfach mit seinem direkten oberen Nachbarn tauscht, und dadurch nach oben kommt.

– First Task First

Dieses Verfahren ist eigentlich mehr für Batchbetrieb, also dem Abarbeiten der

Tasks nacheinander als gleichzeitig gedacht. Es folgt dem Sprichwort „wer zuerst kommt malt zuerst“, sodass ein Programm also den ersten Task ausführt und aber derzeit auch noch andere annimmt. Und nach Beendigung dieses Tasks wird der nächste genommen. Das Verfahren ist unbrauchbar für Systeme mit Mikrokernen, bei denen Treiber und Module auch Tasks darstellen, da diese nicht aufgerufen werden können. Desweiteren scheiden sich die Geister, ob diese Methode überhaupt den Namen „Multitasking“ verdient, da die Tasks nicht gleichzeitig ausgeführt werden, jedoch aber hintereinander.

– Shortest Time Left

Dieses Konzept besagt, dass der Task, der die kleinste Zeit in Anspruch nehmen wird zuerst an die Reihe kommt. Meist bemisst man die Zeit anhand der Größe des Tasks. Jedoch müssen diese Werte ständig aktualisiert werden und es ist schwer für das OS zu sagen, was Code und was Daten sind. Ein Task könnte nur ein Ladeprogramm beinhalten, das den richtigen Task lädt und würde somit als erstes ausgeführt und umgeht damit das System. Somit ist einer der Grundsätze nicht erfüllt.

– Longest Time Left

Dieses Konzept ist exakt das Gegenteil des vorherigen. Es bevorzugt den längsten Task und wieder gelten hier alle Einwände, welche bei „Shortest Time Left“ genannt wurden.

Das waren auch schon die meiner Meinung nach wichtigsten. Es steht jedem frei, weitere dazu zu entwickeln und bestehende zu verfeinern. Ich selber benutze Round Robin ohne jegliche Form von Prioritäten.

9. Taskwechsel

Nun aber zum meiner Meinung kompliziertesten Punkt der Praxis, dem Wechsel selber. Das Implementieren eines Round-Robin-Schedulers sollte nicht alzu schwer gewesen sein, aber der Wechsel hat es in sich, er ist der schwierigste Teil, den ich zu bewältigen hatte, als ich TSS für mein Betriebssystem implementierte.

Zuerst möchte ich schildern, wodurch ein Taskwechsel an sich ausgelöst wird. Es gibt mehrere Möglichkeiten:

1. Ein Far-Jump oder Far-Call bei dem als Selektor ein TSS angegeben wird. Der Offsettingteil wird hierbei ignoriert.
2. Ein IRET nachdem ein Task durch ein TaskGate in der IDT aufgerufen wird, also das NT-Flag gesetzt ist.
3. Ein Far-Jump oder Far-Call auf ein Task-Gate. Auch hier wird der Offsettingteil ignoriert.

Es gibt also 2 Möglichkeiten:

1. Man macht einen Far-Jump auf ein TSS.
2. Man manipuliert das NT-Flag und simuliert den Aufruf als ein Interrupt durch ein TaskGate.

Die zweite Methode habe ich noch nie erfolgreich zum Laufen gebracht und sie ist zweifellos die schwierigere, da man nicht nur die Flags sondern auch BackLink-Felder und Stacks manipulieren muss. Deswegen werde ich nun auch die erste Methode ausgiebig erleutern.

Nun, man möchte sich nun denken „nichts leichter als das“ und schreibt munter in seinen Code „`JMP AX,0x00000000`“, wobei man gedenkt, AX mit der Deskriptornummer zu beladen. Spätestens beim Versuch zu Assemblieren wird man feststellen: das geht nicht. Man muss sich also was anderes ausdenken. Erstmal möchte ich sicherheitshalber das ganze zu diesem ausbauen: „`JMP DWORD AX:0x12345678`“. So wird erzwungen, dass der Assembler auch den kompletten Offsettingteil mit übersetzt und nicht Teile durch Abkürzen unter den Tisch kehrt. Das Offset habe ich zu dieser Zahlenfolge gemacht, da ich es für noch ein Schritt sicherer halte, dass das ganze nicht gekürzt wird. Noch haben wir aber das Problem mit dem AX nicht geklärt. Da dies auf jeden Fall nicht geht wandeln wir auch das um: „`JMP DWORD 0x1234:0x12345678`“.

Dies wird der Assembler zwar problemlos schlucken, aber leider bringt es uns nicht viel. Wie man vielleicht schon merkt setze ich viel darauf, dass der Assembler dies vollständig und ungekürzt in die Opcodes umwandelt, denn das ist unser Angriffspunkt: HexDump. Wir manipulieren einfach den vom Assembler erstellten Maschinencode während der Ausführung. Ein paar Versuche haben ergeben: „JMP DWORD 0x1234:0x12345678“ wird vom Assembler in die Bytes 0xED, 0x78, 0x56, 0x34, 0x12, 0x34 und 0x12 umgewandelt. Wir sehen also, dass die letzten zwei Bytes den Selektor angeben. Wir bleiben also weiterhin dabei, in AX unsere TSS-Deskriptornummer zu haben und bauen uns dann also eine Routine, die die Opcodes manipuliert. Da wir die Bytes in anderen Datentypen auslesen als sie normalerweise abgelegt sind kommt es zu derartigen Verdrehungen, da die CPU andere Zahlendarstellungsverfahren benutzt, als der Mensch und die Zahlen konvertiert werden (näheres in meinem Tutorial „Datenveränderungen im RAM“). Wir können also getrost einfach ein Word an die Stelle mit dem Selektor schreiben, der sich 5 Bytes hinter Befehlsbeginn befindet. Den Befehlsbeginn finden wir anhand eines Labels und schon haben wir alles zusammen um folgenden Code zu bauen:

```
PUSH EDI
MOV EDI, tss_switch_label ; Offset laden
ADD EDI, 0x00000005 ; zum Operanten navigieren
STOSW ; neuen Wert eintragen
tss_switch_label:
JMP DWORD 0x1234:0x12345678 ; springen!
POP EDI
```

Der Code verlangt lediglich, dass sich in AX die TSS-Deskriptornummer des Tasks befindet, zu dem gewechselt werden soll. Alles andere macht die CPU. Somit dürfte das Wechseln auch kein Problem mehr sein.

10. Implementierung und Praxis

Tja, wir haben jetzt alles wichtige besprochen, Zeit, zusammen mal einen Code zu coden, den ihr in euer TimerIRQ einbauen könnt, und der für euch den Taskwechsel macht. Zuerst einmal brauchen wir den Scheduler. Ich gehe davon aus, dass die Task-Tabelle an der Adresse 0x00005000 liegt. Die Größe wird anhand einer Variable angegeben, die das Offset des Endes beinhaltet, welche „tasking_end“ heißt und vom Typ Double Word ist. Außerdem existiert eine Variable mit dem Namen „tasking_current“, welche ebenfalls vom Typ Double Word ist und den aktuellen Punkt in der Tabelle festhält.

In unserem Code bauen wir reines Round Robin, wir brauchen also Task-Tabellen-Einträge nach der Form, welche unter „4. Task-Tabelle“ gezeigt ist. Ein Eintrag ist also 6 Bytes groß.

Coden wir nun einfach mal den Scheduler. Da wir am Anfang des IRQs keinen Dispatcher brauchen, können wir gleich mit dem Scheduler anfangen.

```
ADD DWORD [tasking_current], 0x00000006 ; Werte des
; aktuellen Tasks überspringen
MOV ESI, [tasking_end] ; Task-Tabellen-Ende laden
CMP ESI, [tasking_current] ; mit Position des zu ladenden
; Tasks vergleichen
JNE scheduler_no_resetting ; falls nicht gleich, also
; das Ende noch nicht erreicht ist springen
MOV DWORD [tasking_current], 0x00005000 ; sonst auf den
; Anfang zurücksetzen
scheduler_no_resetting:
; hier haben wir in „tasking_current“ die Adresse der
; Daten des zu ladenden Tasks
```

Da wir in diesem Scheduler ein Register verändern und dies natürlich nicht sein darf, müssen wir diese (nur ESI) noch auf dem Stack sichern:

```

PUSH ESI
ADD DWORD [tasking_current],0x00000006 ;Werte des
    ; aktuellen Tasks überspringen
MOV ESI,[tasking_end] ; ask-Tabellen-Ende laden
CMP ESI,[tasking_current] ;mit Position des zu ladenden
    ; Tasks vergleichen
JNE scheduler_no_resetting ;falls nicht gleich, also
    ; das Ende noch nicht erreicht ist spingen
MOV DWORD [tasking_current],0x00005000 ;sonst auf den
    ; Anfang zurücksetzen
scheduler_no_resetting:
POP ESI
    ; hier haben wir in „tasking_current“ die Adresse der
    ; Daten des zu ladenden Tasks

```

Nun müssen wir aus der Tasktabelle die TSS-Deskriptornummer des zu landenden Tasks raussuchen und in AX bekommen:

```

PUSH ESI
ADD DWORD [tasking_current],0x00000006 ;Werte des
    ; aktuellen Tasks überspringen
MOV ESI,[tasking_end] ; ask-Tabellen-Ende laden
CMP ESI,[tasking_current] ;mit Position des zu ladenden
    ; Tasks vergleichen
JNE scheduler_no_resetting ;falls nicht gleich, also
    ; das Ende noch nicht erreicht ist spingen
MOV DWORD [tasking_current],0x00005000 ;sonst auf den
    ; Anfang zurücksetzen
scheduler_no_resetting:
POP ESI
    ; hier haben wir in „tasking_current“ die Adresse der
    ; Daten des zu ladenden Tasks
MOV ESI,[tasking_current] ; Offset laden
ADD ESI,0x00000002 ; zur Deskriptornummer navigieren
LODSW ; Nummer in AX laden

```

Da wir wieder die Register verändern müssen wir auch das Stacksichern verändern:

```

PUSH EAX
PUSH ESI
ADD DWORD [tasking_current],0x00000006 ;Werte des
    ; aktuellen Tasks überspringen
MOV ESI,[tasking_end] ; ask-Tabellen-Ende laden
CMP ESI,[tasking_current] ;mit Position des zu ladenden
    ; Tasks vergleichen
JNE scheduler_no_resetting ;falls nicht gleich, also
    ; das Ende noch nicht erreicht ist spingen
MOV DWORD [tasking_current],0x00005000 ;sonst auf den
    ; Anfang zurücksetzen
scheduler_no_resetting:
    ; hier haben wir in „tasking_current“ die Adresse der
    ; Daten des zu ladenden Tasks
MOV ESI,[tasking_current] ; Offset laden
ADD ESI,0x00000002 ; zur Deskriptornummer navigieren
LODSW ; Nummer in AX laden
POP ESI
POP EAX

```

Nun müssen wir nurnoch unseren Dispatcher dahinter hängen und die Stacksicherung wieder verschieben und schon ist der Code fertig:

```

PUSH EAX

```

```

PUSH ESI
ADD DWORD [tasking_current],0x00000006 ;Werte des
    ; aktuellen Tasks überspringen
MOV ESI,[tasking_end] ; ask-Tabellen-Ende laden
CMP ESI,[tasking_current] ;mit Position des zu ladenden
    ; Tasks vergleichen
JNE scheduler_no_resetting ;falls nicht gleich, also
    ; das Ende noch nicht erreicht ist spingen
MOV DWORD [tasking_current],0x00005000 ;sonst auf den
    ; Anfang zurücksetzen
scheduler_no_resetting:
    ; hier haben wir in „tasking_current“ die Adresse der
    ; Daten des zu ladenden Tasks
MOV ESI,[tasking_current] ; Offset laden
ADD ESI,0x00000002 ; zur Deskriptornummer navigieren
LODSW ; Nummer in AX laden
POP ESI
PUSH EDI
MOV EDI, tss_switch_label ; Offset laden
ADD EDI,0x00000005 ; zum Operanten navigieren
STOSW ; neuen Wert eintragen
tss_switch_label:
JMP DWORD 0x1234:0x12345678 ; springen!
POP EDI
POP EAX

```

So, fertig ist der Praxisteil. Ich bitte alle Leser, diesen Code nicht einfach nur blind zu kopieren, denn er lässt sich sicher nicht fehlerlos in das System integrieren, und so schwer ist das Implementieren nicht. Also selber versuchen und bei Fragen mich einfach kontaktieren.

Damit dieser Code richtig geht brauchen wir aber auch einen Task, der gerade läuft. Angezeigt wird dieser Task durch das TR, das TaskRegister. Dieses beinhaltet einen Selektor auf den Deskriptor des aktuellen TSS. Um der CPU vorzugaukeln, welcher Task gerade aktiv ist, benutzen wir den Befehl „LTR Operant“. Als Operant wird die Deskriptornummer des Tasks, der gerade aktiv ist angegeben. Beim Wechsel aktualisiert die CPU das TR dann automatisch. Man braucht den Befehl also nur einmal.

11.Tipps & Tricks

Jetzt habe ich zu Schluss noch ein paar Tipps zum Programmieren von Multitasking und zum Programmieren allgemein.

1. Programmiere auf Sicherheit! Es geht nicht um einen Geschwindigkeitsrekord, sondern um ein stabiles Betriebssystem. Deswegen beachte und bedenke jeden Fehler, der auftreten könnte und richte Fehlerbehandlungen ein. Du musst sie nicht gleich rein Implementieren, aber schaffe (wenn auch nur leere) Funktionen dafür, die du später erweitern kannst. Somit hast du ein sicheres Gerüst und gerätst nicht in die Falle, dass ein Code nicht geht, weil ein unerwarteter Fehler aufgetreten ist.
2. Schreibe nicht alles ab! Code den Code selber. Es wird dir helfen, das Zeug zu verstehen und eigener Code lässt sich besser ins System integrieren.
3. Entscheide dich fest für ein System und mach keine Abstriche. Versuche deine Gedanken durchzusetzen und gib' nicht auf, blos, weil was nicht klappt.
4. Achte auch die Optimierung deines Codes. Schon kleine Teile könnte vieles verschnellern und dein System sollte das Minimum an Leistung verbrauchen.
5. Halte alles Modular, sprich baue auch für deinen Scheduler eine Funktion. Das wird es später erleichtern, die Funktion auszutauschen, wenn du eine andere Strategie verwenden willst.
6. Und als letztes: Kommentiere deinen Code gut. Gerade in Assembler ist das sehr

wichtig.

Tja, das waren auch schon meine Tipps.

12.Schluss

Wir sind nun schon am Ende dieses Tutorials angelangt. Es ist etwas lang geworden, warscheinlich deshalb, dass Multitasking nunmal ein komplexes Thema ist, dass man nicht so einfach erklären kann. Ich hoffe es hat Spaß gemacht, ein bisschen in die Welt des Multitasking einzutauchen und davon zu hören. Bei Fragen kann ich nur wieder ermuntern, sich bei mir zu melden. Ich helfe, so gut ich kann. Ich bin unter E-Mail joachim_neu@web.de, ICQ# 124-390-343 und MSN joachim_neu@web.de erreichbar. Ich habe auch eine Internetseite, bei der ich mich über ein Besuch sehr freuen würde: www.joachim-neu.de. Viele Fragen kann man auch im Forum des deutschen Magazins für Betriebssystemprogrammierung „Lowlevel“ stellen und bekommt schnell gute Antworten. Also schau' vorbei unter: www.lowlevel.net.tc. Viel Spaß beim Betriebssystemprogrammieren noch, Joachim Neu alias „joachim_neu“.