Fachhochschule Würzburg-Schweinfurt

Wintersemester 2007/2008

Diplomfachprüfung im Fach

Prozessdatenverarbeitung II

(Prof. Dr.-Ing. Ludwig Eckert)

Datum: 31.01.2008, 08.00 Uhr, Raum 0437

Dauer: 90 Minuten Erreichte Punktzahl: .....

Note: .....

Hilfsmittel: Ohne schriftliche Unterlagen, ohne Mobiltelefone, Taschenrechner erlaubt.

Vorname: ..... Name: .....

Matrikelnr: ..... eMail-Adresse: .....

Erstprüfer: .....

Zweitprüfer: .....

Wichtige Hinweise:

Bitte tragen Sie alle Antworten auf den folgenden Seiten direkt im Anschluss an die Aufgabentexte ein.

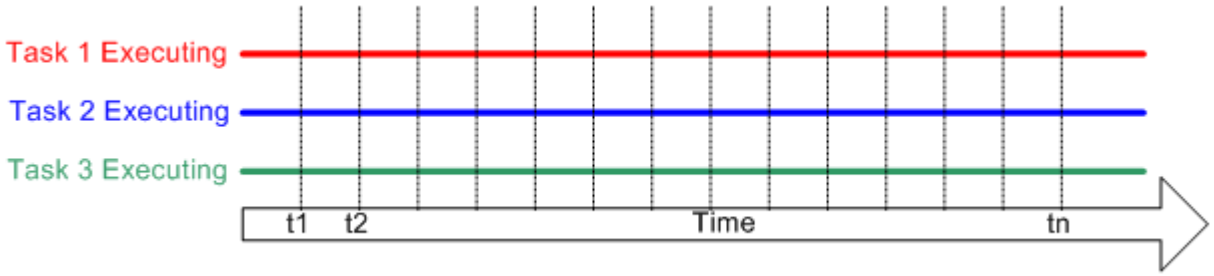
Ergebnisse auf Zusatzblättern werden nur in begründeten Ausnahmefällen gewertet.

Aufgabe 1: Echtzeitbetriebssysteme
Erläutern Sie den Unterschied zwischen „harten“ und „weichen“ Echtzeitsystemen hinsichtlich der zeitlichen Anforderungen?

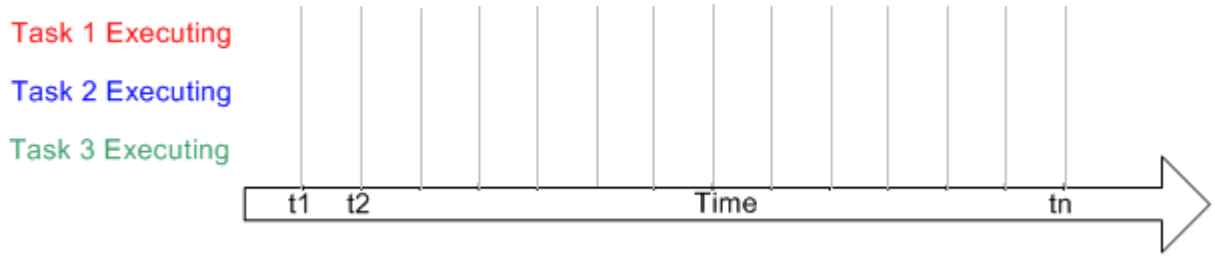
Aufgabe 2: RTOS
Welche Vorzüge hat der Einsatz eines RTOS Kernels für die Programmierung von Embedded Systemen? Nennen Sie fünf Vorteile.

Aufgabe 3: RTOS
Wie werden zwei ablaufbereite Tasks gleicher Priorität von einem RTOS-Scheduler behandelt? (Ein-Prozessor System wird vorausgesetzt)

Aufgabe 4: RTOS
In einem RTOS basierenden Embedded System sollen drei Tasks unendlich lange laufen. Alle 3 Tasks haben die gleiche Priorität.



Zeichnen Sie in das nachstehende Diagramm ein, wie sich die Rechenzeit des Ein-Prozessor Systems auf die einzelnen Tasks verteilt.



Aufgabe 5: RTOS

a) Was versteht man unter einem „Idle“-Task?

b) Wann wird dieser Task von dem RTOS-Scheduler aufgerufen?

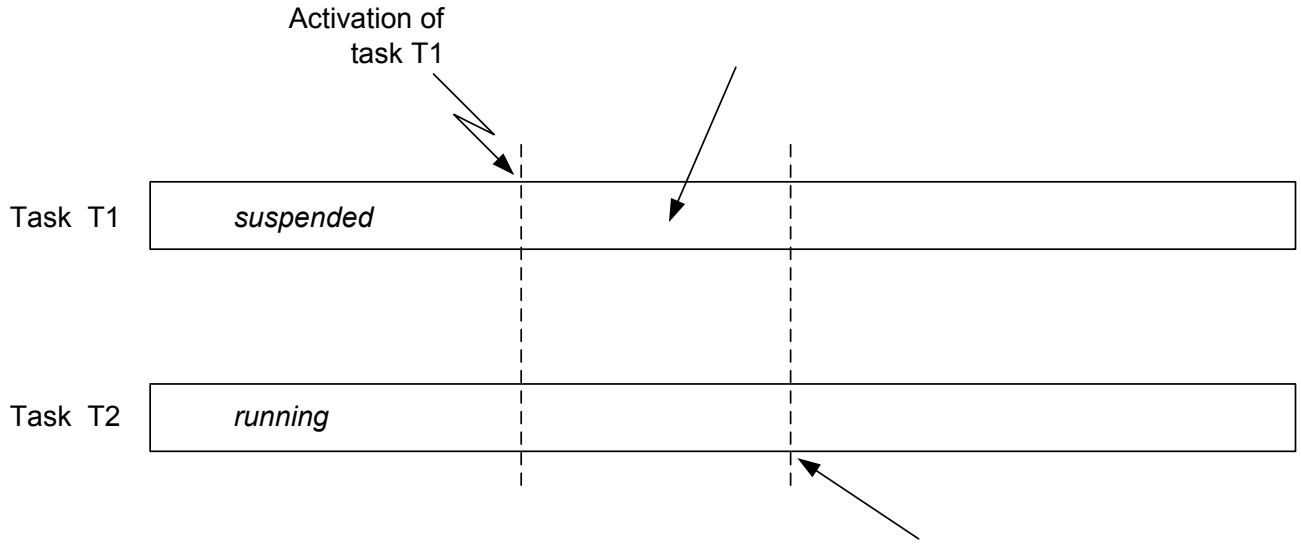
c) Welche Möglichkeiten gibt es in diesem Zusammenhang den Energieverbrauch eines Embedded Systems zu reduzieren? (Begründung)

Aufgabe 6: RTOS

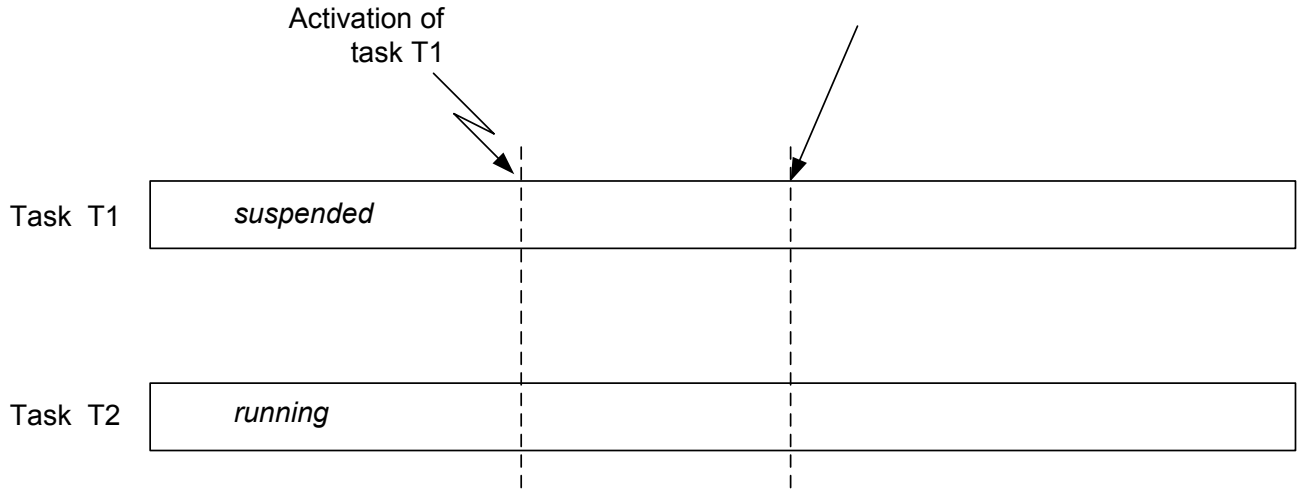
Erläutern Sie den Unterschied im RTOS Verhalten zwischen pre-emptive and cooperative Multitasking anhand zweier Tasks in einem Beispiel. Task T1 hat eine höhere Priorität als Task T2.

Vervollständigen Sie die nachstehende Darstellung und tragen Sie die Taskzustände (suspended, ready, running) direkt in das Bild ein und beschriften Sie die Pfeile.

Cooperative Multitasking



Pre-emptive Multitasking



Aufgabe 7: Echtzeitbetriebssysteme
Nennen Sie fünf Schedulingstrategien.

Aufgabe 8: Echtzeitbetriebssysteme
Was versteht man unter Priority Inversion und welches RTOS-Problem wird hierdurch gelöst? Erläutern Sie den Begriff an einem Beispiel.

Aufgabe 9: Taskmanagement
Aufgabe 9.1
Auf welche Weise gelangt ein Anwenderprozess in den Zustand „delayed“ und aus diesem Zustand wieder heraus?

Aufgabe 9.2:
Auf welche Weise gelangt ein Anwenderprozess in den Zustand „blockiert“ und aus diesem Zustand wieder heraus? Nennen Sie drei mögliche Gründe dafür.

Aufgabe 10:
Nachfolgend ist das zeitliche Sollverhalten von 4 Rechenprozessen (RP1-4) dargestellt, wobei der RP4 die höchste und RP1 die niedrigste Priorität aufweist.

Aufgabe 10.1:
Tragen Sie in das nachstehende Bild die tatsächliche Abarbeitung der Rechenprozesse ein (bis zum normierten Zeitpunkt t/T = 12).

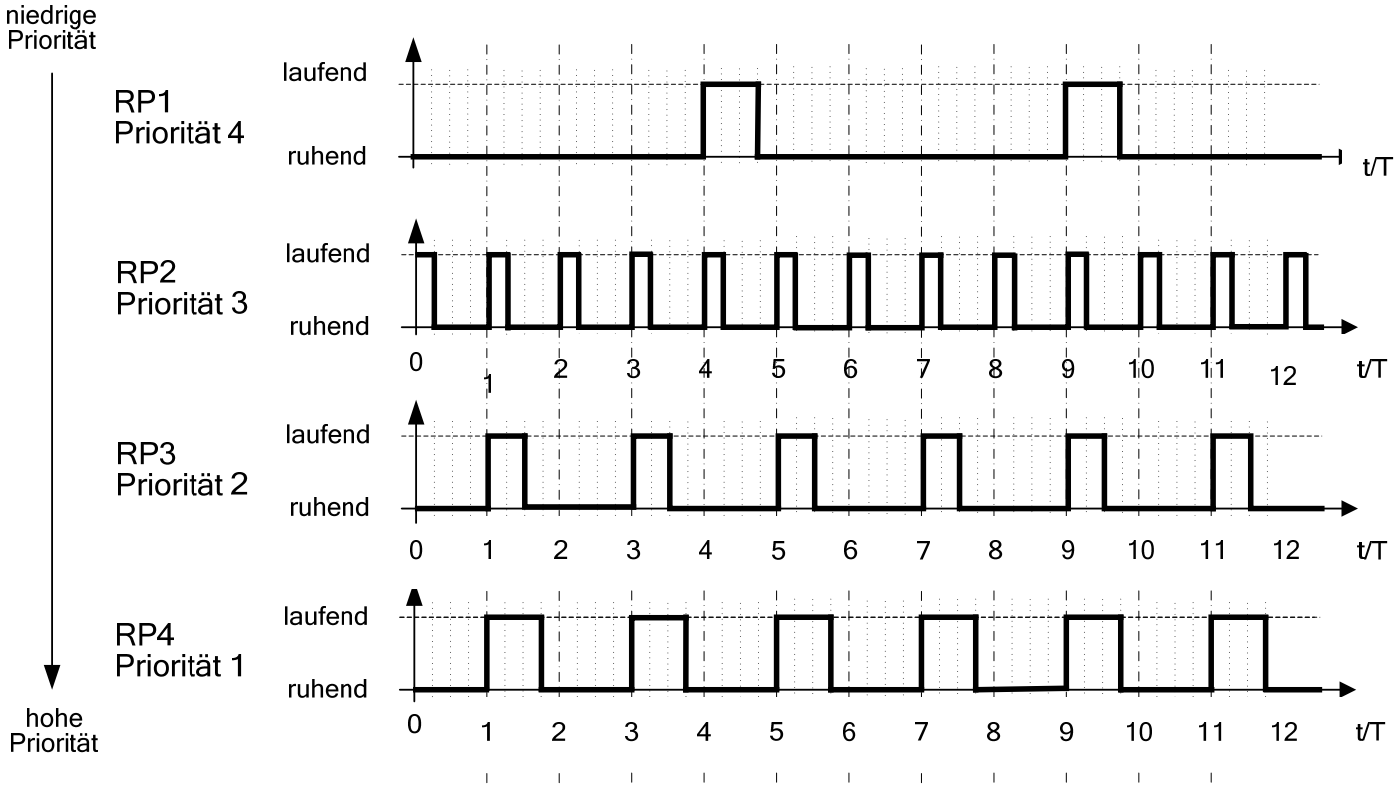


Bild: Zeitliches Sollverhalten der Rechenprozesse

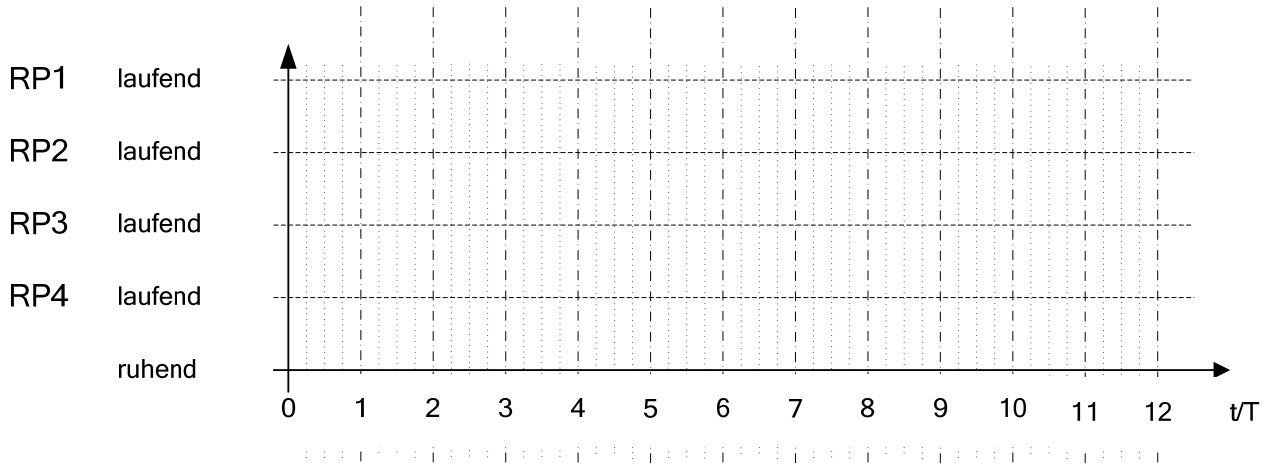


Bild: Tatsächlicher zeitlicher Ablauf der Rechenprozesse

Aufgabe 10.2:
Ein Prozess gilt als rechtzeitig ausgeführt, wenn das Rechenergebnis noch vor Beginn der nächsten Periode vorliegt. Wie beurteilen Sie unter diesem Gesichtspunkt die Rechtzeitigkeit der einzelnen Rechenprozesse RP1 bis RP4?

RP1:

RP2:

RP3:

RP4:

Aufgabe 10.3:
Berechnen Sie den Auslastungsgrad des Prozessors unter der Annahme, dass alle Prozesse auf demselben Prozessor abgearbeitet werden.

Aufgabe 10.4:
Tragen Sie den zeitlichen Verlauf der Taskzustände der Rechenprozesse RP1, RP2 und RP3 in die nachstehenden Diagramme ein.

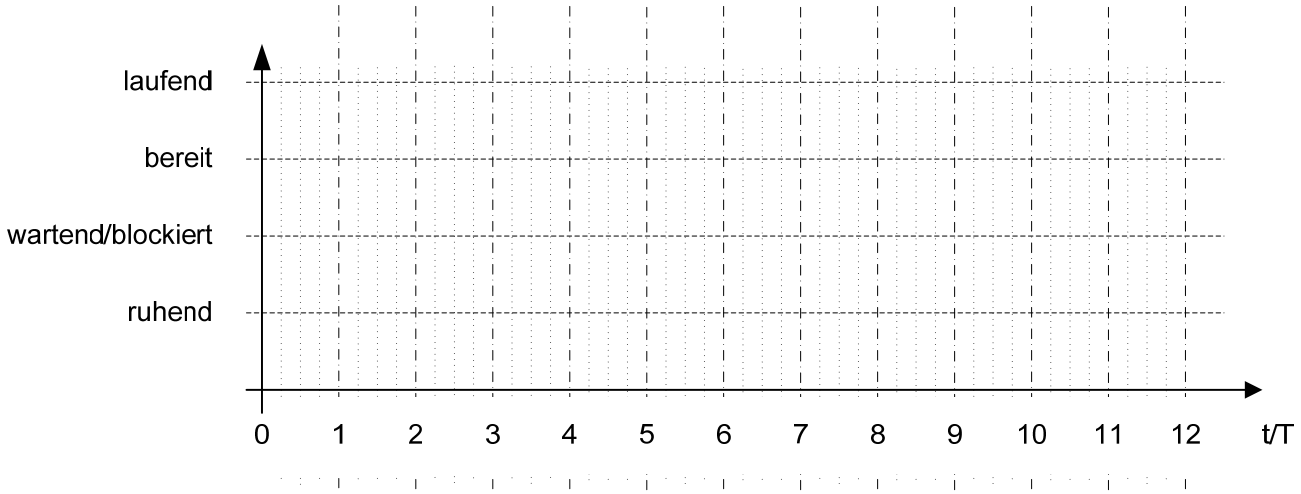


Bild: Taskzustände des Rechenprozesses RP1

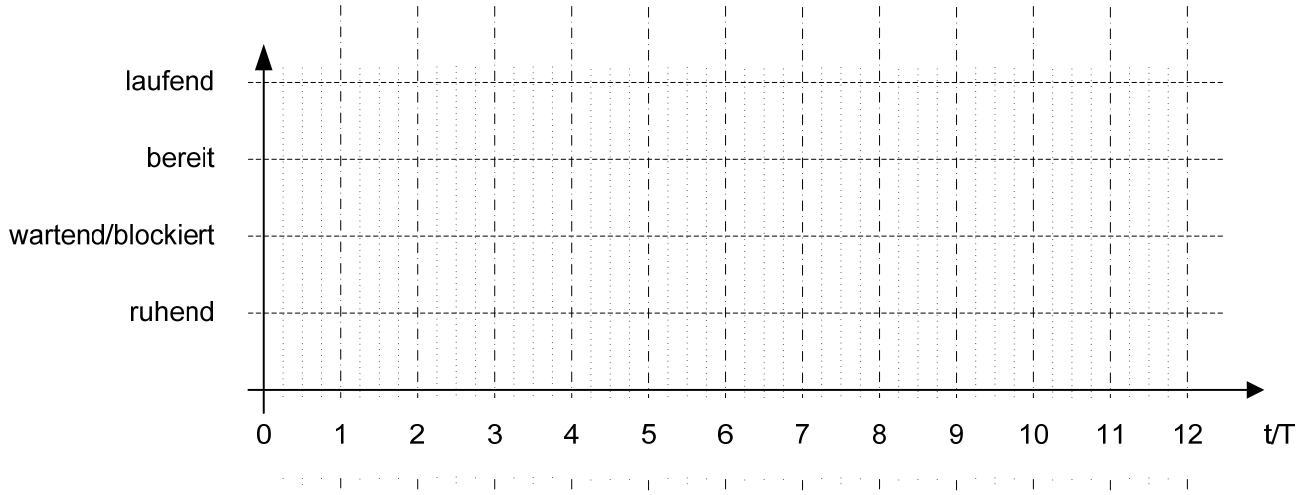


Bild: Taskzustände des Rechenprozesses RP2

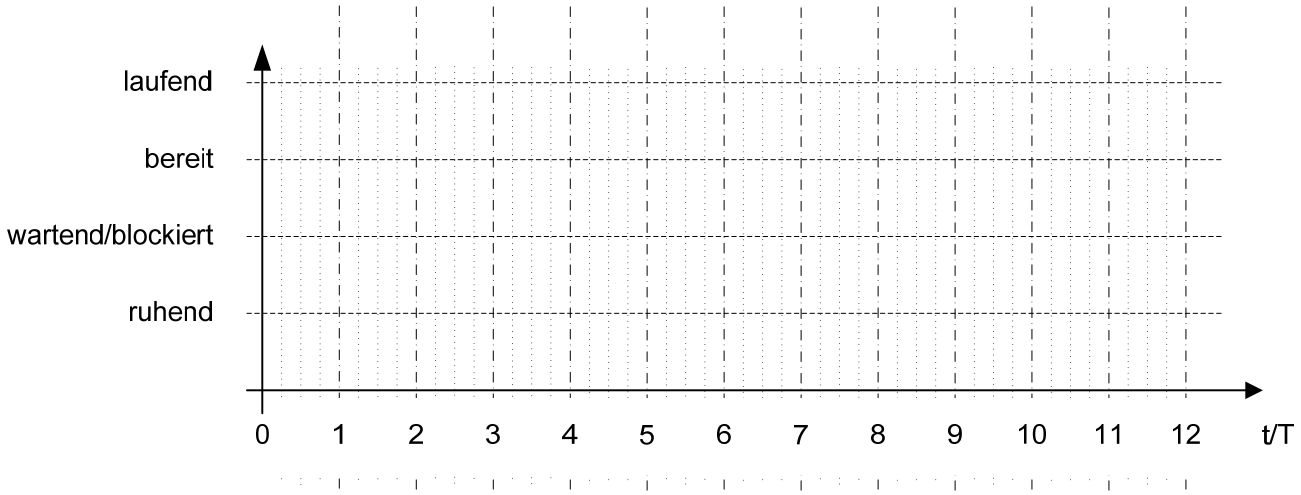


Bild: Taskzustände des Rechenprozesses RP3

Aufgabe 11:
Gegeben ist das nachstehende VxWorks-Programm. Bitte analysieren Sie dieses und beantworten Sie die nachstehenden Fragen.

```
// includes
#include "vxWorks.h"
#include "taskLib.h"
#include "semLib.h"
#include "stdio.h"

// function prototypes
void taskOne(void);
void taskTwo(void);

// globals
#define ITER 10
SEM_ID semBinary;
int global = 0;


void binary(void)
{ int taskIdOne, taskIdTwo;
  // create semaphore with semaphore available and queue tasks on FIFO basis
  semBinary = semBCreate(SEM_Q_FIFO, SEM_FULL);

  // Note 1: lock the semaphore for scheduling purposes
  semTake(semBinary,WAIT_FOREVER);            // take the semaphore

  // spawn the two tasks
  taskIdOne = taskSpawn("t1",90,0x100,2000,(FUNCPTR)taskOne,0,0,0,0,0,0,0,0,0,0);
  taskIdTwo = taskSpawn("t2",90,0x100,2000,(FUNCPTR)taskTwo,0,0,0,0,0,0,0,0,0,0);
}

void taskOne(void)
{ int i;
  for (i=0; i < ITER; i++)
            {
            semTake(semBinary,WAIT_FOREVER); // wait indefinitely for semaphore
            printf("I am taskOne and global = %d.....\n", ++global);
            semGive(semBinary);             // give up semaphore
            }
}

void taskTwo(void)
{
  int i;
  semGive(semBinary);      // Note 2: give up semaphore
  for (i=0; i < ITER; i++)
            {
             semTake(semBinary,WAIT_FOREVER);  // wait indefinitely for semaphore
             printf("I am taskTwo and global = %d--.----\n", --global);
             semGive(semBinary);             // give up semaphore
            }
}
```

Aufgabe 11.1:
Protokollieren Sie den Wert der globalen Variablen *global* über den gesamten Programmablauf.

Aufgabe 11.2:
Was ist der offensichtliche Zweck des Programms?

Aufgabe 11.3:

Wie würde sich die Variable *global* verändern, wenn alle *semGive()* und s*emTake()* Befehle entfernt werden würden?

Annahme: Der Scheduler arbeitet prioritätsorientiert, ein „Round Robin“-Verfahren ist nicht implementiert.

Aufgabe 12: Tasksynchronisation

Im nachstehenden Programm ist die Anordnung von Semaphoroperationen am Anfang und am Ende der Tasks dargestellt. Ermitteln Sie, ob und in welcher Reihenfolge diese Tasks bei einer Initialisierung nach 10a), 10b) und 10c) ablaufen. SA, SB und SC stellen Counting Semaphoren dar.

```
Task_A                  Task_B                  Task_C
{                       {                       {
 SemTake(SA)             SemTake (SB)            SemTake(SC)
 SemTake(SA)             .                       SemTake(SC)
 SemTake(SA)             .                       SemTake(SC)
        .                .                       .
        .                .                       .
        .                .                       .
        .                .                       .
        .                .                       .
        .                SemGive(SC)             SemGive(SB)
 SemGive(SB)             SemGive(SA)             SemGive(SB)
}                       }                       }
```

Bild: Anordnung der Semaphoroperationen in den Tasks

10a)

	Werte der Semaphorvariablen			Reihenfolge der Tasks
	SA	SB	SC	
Anfangswerte	2	0	2	

10b)

	Werte der Semaphorvariablen			Reihenfolge der Tasks
	SA	SB	SC	
Anfangswerte	3	0	2	

10c)

	Werte der Semaphorvariablen			Reihenfolge der Tasks
	SA	SB	SC	
Anfangswerte	0	0	3	

Anhang: Befehle zur Tasksynchronisation und -kommunikation

SEM_ID = semBCreate(int *options*, SEM_B_STATE *initialState*)

// Erzeugung und Initialisierung einer binären Semaphore
// SEM_ID: Rückgabewert: Semaphor ID oder NULL, falls die Semaphore nicht erzeugt werden konnte
// options: SEM_Q_PRIORITY: Warteschlange nach Priorität und FIFO-Prinzip,
// SEM_Q_FIFO: Warteschlange nur nach FIFO-Prinzip.
// initalState: SEM_EMPTY oder SEM_FULL.
// Hinweis: Der Semaphor ist nach Aufruf arbeitsbereit.

SEM_ID = semMCreate(int *options*, SEM_B_STATE *initialState*)

// Erzeugung und Initialisierung einer Mutual Exclusion Semaphore

SEM_ID = semCCreate(int *options*, SEM_B_STATE *initialState*)

// Erzeugung einer Couting Semaphore

STATUS = semGive(SEM_ID *semId*)

// „signalisiert“ die Semaphore semId
// STATUS: Rückgabewert: OK oder ERROR, wenn Fehler bei der Signalisierung

STATUS = semTake(SEM_ID *semId*, int *timeout*)

// „nimmt“ die Semaphore semId
// STATUS: Rückgabewert: OK oder ERROR, wenn Semaphore semId nicht vorhanden ist oder wenn
// eine Wartezeitüberschreitung aufgetreten ist.
// timeout: NOWAIT, WAIT_FOREVER oder Anzahl Ticks warten
// Hinweis: Diese Funktion kann nicht in einer ISR aufgerufen werden.

STATUS = semDelete(SEM_ID *semId*)

// Semaphore semId wird durch semDelete() gelöscht.
// STATUS: Rückgabewert: OK oder ERROR
// Hinweis: Der Semaphor wird gelöscht.

STATUS = semFlush(*SEM_ID semId*)

// STATUS: Rückgabewert: OK oder ERROR
// Hinweis: Alle Tasks, die auf die Semaphore warten, werden in den Ready-Zustand versetzt.

TaskId = taskSpawn (*name, priority, options, stacksize, entrypoint, arg1, .. arg10*)

// Task kreieren und aktivieren
// TaskId: ist die vom System zugewiesene TaskID
// name: Name der Task, bestehend aus ASCI Zeichen, z. B. "tMeine_Task"
// priority: Priorität in VxWorks wählbar zwischen 0 (höchste) bis 255 (niedrigste) Priorität
// options: Optionen z. B. für das "Task-Debugging"
// stacksize: Größe des Stackspeichers für einen Task in Bytes
// entrypoint: Hier wird der Name der C-Funktion eingetragen, die im Taskkontext ablaufen soll.
// arguments: Diese Argumente sind die C-Funktionsübergabeparameter der entrypoint-Funktion.

Funktionen zur Taskkommunikation

MSG_Q_ID = msgQCreate (int maxMsgs, int maxMsgLength, int options)

// int maxMsgs Maximale Anzahl von Nachrichten in der Queue
// int maxMsgLength, Anzahl Bytes in einer Nachricht
// int options Sortieralgorithmus, siehe übernächste Zeile
// MSG_Q_FIFO Queue wird nach dem FIFO-Prinzip organisiert
// MSG_Q_PRIORITY Nachrichten werden nach der Priorität der
// sendenden Task eingeordnet.
// Rückgabewert: ID der Queue (Pointer) oder NULL
// Hinweis: Funktion kann nicht in einer ISR verwendet werden.

MSG_Q_ID = msgQDelete (MSG_Q_ID msgQId)

// MSG_Q_ID msgQIdID der zu löschenden Queue
// Rückgabewert OK oder ERROR
// Hinweis: Funktion kann nicht in einer ISR verwendet werden.

Eine Task sendet bzw. empfängt Nachrichten einer bestimmten Message-Queue mittels des Aufrufes von Funktionen.

STATUS = msgQSend (MSG_Q_ID msgQId, char* buffer, UINT nBytes, int timeout, int priority)

// MSG Q_ID msgQId ID der zu benutzenden Queue
// char* buffer Zu sendende Nachricht
// UINT nBytes Länge der Nachricht
// int timeout
// NO_WAIT: Kein Warten, auch wenn die Queue voll ist und die Nachricht nicht gesendet werden kann.
// WAIT_FOREVER: Warten bis Senden möglich ist
// Mit Timeout: Anzahl von Ticks auf Sendebereitschaft der Queue warten.
// int priority
// MSG_PRI_NORMAL: Nachricht an das Ende der Queue hängen
// MSG_PRI_URGENT: Nachricht am Anfang der Queue eintragen
// Rückgabewert: OK oder ERROR, Error wird gesetzt.

STATUS = msgQReceive (MSG_Q_ID msgQId, char* buffer, int timeout)

// MSG_Q_ID msgQId ID der zu benutzenden Queue
// char* buffer Puffer, in den Nachricht geschrieben wird
// UINT maxNBytes Pufferlänge
// int timeout
// Timeout:
// NO_WAIT Kein Warten, auch wenn die Queue leer ist
// WAIT_FOREVER Warten bis Empfang möglich ist.
// Mit *timeout* Anzahl von Ticks auf Empfang warten.
// Rückgabewert; Anzahl der empfangenen Bytes oder ERROR, errno wird gesetzt.

// Hinweis: Funktion kann nicht in einer ISR verwendet werden.