3. Übung: speisende Philosophen **Realtime Systems**

Abgabe: 9. Juni 2008 (für die Gruppen 1 und 3)
16. Juni 2008 (für die Gruppe 2)

Schreiben Sie ein Programm zur Simulation des Problems der speisenden Philosophen. Bilden Sie die Philosophen auf Threads und die Stäbchen sowie den Diener auf Semaphore ab.

Verwenden Sie für die Anzahl der Philosophen ein Präprozessor-Makro, um sie leicht verändern zu können.

Denken Sie daran, daß es eine maximale Zahl von gleichzeitig speisenden Philosophen gibt, bei der keine Gefahr einer Verklemmung besteht. Sichern Sie dies über einen weiteren Semaphor (`Diener`).

Simulieren Sie das "Essen" und das "Denken" jeweils durch eine neue Zufallszeit (im Bereich von einigen 10 ms unter Verwendung von `usleep(...)`), wobei die Zeit für das "Denken" länger sein sollte als die für das "Essen".

Sehen Sie die geordnete Beendigung des Programms, z.B. auf eine Tastatureingabe hin, vor (z.B. unter Verwendung von `getchar()` und einer globalen Variablen `running`).

Geben Sie bei der Beendigung des Programmes aus, wie oft jeder Philosoph gegessen hat.

Das nachfolgende Beispiel zeigt die Verwendung von Semaphoren zur Synchronisation zwischen dem main-Thread und einem Kind-Thread. Es stellt ***keinen*** Lösungsvorschlag für diese Aufgabe dar. Diesen Quelltext finden Sie auch unter
`/pst/uebungen/doz/buchholz/rts/thread/thread_sem.c`.

```
/*****************************************************************************/
/*        Beispiel fuer den Einsatz von Threads und Semaphoren:              */
/*                                                                           */
/* Complilieren und Linken mit:                                              */
/*      gcc -Wall -D_REENTRANT thread_sem.c -lpthread -o thread_sem           */
/*****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define MAL       10000
#define INTER_THREAD    0

/* Definition von Semaphoren als !globale! Variablen.                        */

sem_t go_on1, go_on2;


/* Diese Funktion enthaelt den Programmcode, der als Thread ausgefuehrt wird.*/
```

```
void myThread(int param)
{
      int i;

      for (i=0; i<MAL; i++) {
            if (sem_wait(&go_on1) < 0) {
                  perror("sem_wait");
                  pthread_exit(0);
            }

            if (sem_post(&go_on2) < 0) {
                  perror ("sem_post");
                  pthread_exit(0);
            }
      }
}



int main()
{
      pthread_t tid;
      long status;
      int i;



/* sem_int(...) erzeugt den Semaphor go_on1 und initialisiert ihn mit 0.     */

      if (sem_init(&go_on1, INTER_THREAD, 0) < 0) {
            perror("sem_init");
            return(1);
      }

      if (sem_init(&go_on2, INTER_THREAD, 0) < 0) {
            perror("sem_init");
            return(1);
      }

/* pthread_create(...) erzeugt einen neuen Thread, dessen Programmzaehler    */
/* auf die Startadresse der Funktion myThread gesetzt wird. Als Parameter    */
/* kann ein vier-Byte langer Wert uebergeben werden (hier 123 aber sonst     */
/* auch oft die Adresse einer Struktur oder eines Feldes.                    */
/* tid enthaelt nach erfolgreicher Ausfuehrung die Thread ID des erzeugten   */
/* Threads.                                                                  */

      if (pthread_create(&tid, NULL, (void *)myThread, (void *)0) == -1) {
            perror("create");
            return(1);
      }
```

```
/* Der Aufruf sem_post(&go_on1) entspricht der V(...)-Funktion und          */
/* korrespondiert zum Aufruf von sem_wait(&go_on1...) in myThread.          */

      for (i=0; i<MAL; i++) {
            if (sem_post(&go_on1) < 0) {
                  perror ("sem_post");
                  return(1);
            }

            if (sem_wait(&go_on2) < 0) {
                  perror("sem_wait");
                  return(1);
            }
      }

/* pthread_join(tid, &status) blockiert solange, bis der Thread mit der     */
/* Thread-ID tid beendet wurde. In status wird der Status der Beendigung    */
/* mitgeteilt.                                                              */

      if (pthread_join(tid, (void **)&status) == -1) {
            perror("join");
            return(1);
      }

      return(0);
}
```