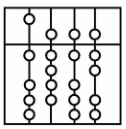
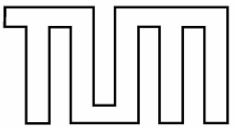


INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN



Informatik VI: Robotics and Embedded Systems
Prof. Dr. A. Knoll

Prozessrechner-Praktikum
Echtzeitsysteme

Aufgaben 0-3

C. Buckl buckl@in.tum.de
G. Schrott schrott@in.tum.de

Sommersemester 2006

1 Praktikumsaufgaben

1.1 Übersicht über die Aufgaben

0. Einführung in C unter VxWorks
1. Multitasking und Interprozesskommunikation
2. Zyklisch ablaufende Prozesse
3. Erzeuger-Verbraucher-Problem
4. Kugelfall-Versuch
5. Aufzugssteuerung
6. Murmelbahn

1.2 Erfolgreiche Teilnahme am Praktikum

Folgende Leistungen sind Voraussetzung für den Erwerb des Praktikumscheins:

a) Die Aufgaben 0-6 müssen bearbeitet werden.

b) **Anwesenheit an allen Praktikumsnachmittagen** ist nötig.

c) Die Lösung zu Aufgabe 5 und 6 ist vorzuführen.

d) Am Ende des Praktikums findet gruppenweise ein **mündliches Kolloquium** statt.

e) **Eine Woche vor Praktikumsende ist pro Gruppe eine Ausarbeitung abzugeben**, die für jede der bearbeiteten Aufgaben ein auf DIN A4-Format zugeschnittenes Programm-Listing und eine Dokumentation der einzelnen Programme enthält.

2 Aufgabe 0: Einführung in C unter VxWorks

2.1.1 Ziel

C ist eine weit verbreitete Programmiersprache, die sich durch Bibliotheken nahezu beliebig erweitern lässt. Die erste Aufgabe soll Ihnen helfen sich in C einzuarbeiten und einige Möglichkeiten und Besonderheiten, die bei der C-Programmierung unter VxWorks zu beachten sind, nahe bringen.
Zudem bietet die Aufgabe eine Einleitung in die Benutzung der POSIX-Bibliothek, die sie im Rahmen des Praktikums benutzen werden und insbesondere in die Threadprogrammierung.
Eine Übersicht über die POSIX-Funktionalität finden sie auf der Praktikumshomepage.

2.1.2 Aufgabe

Verbessern Sie das fehlerhafte C-Programm *aufgabe0.c*, das sich im Verzeichnis *Praktikum* auf dem Rechners *atknoll41* befindet sowie auf der nächsten Manualseite abgedruckt ist. Arbeiten Sie mit einer Kopie des Programms *aufgabe0.c* in Ihrem Projekt-Verzeichnis auf dem Rechner *sunknoll1*. Führen Sie das korrigierte Programm am Simulator sowie am Zielrechner aus. Die Aufrufreihenfolge der Prozesse hängt unter anderem von deren Priorität ab. Betrachten Sie deshalb bei dieser Aufgabe auch das Zusammenspiel der Prozesse unter Berücksichtigung der jeweils vergebenen Prioritäten!
Erläutern Sie zudem die einzelnen Programmschritte und fassen Sie die gefundenen Fehler zusammen. Senden Sie die verbesserte Version, die Programmbeschreibung und die Liste der Fehler an den Praktikumsbetreuer.

2.1.3 Implementierungshinweise

Zum Thema C-Programmierung gibt es eine fast unüberschaubare Vielfalt von Fachbüchern. Besonders empfehlenswert zum Erlernen von C ist die Lektüre von Kernighans & Ritchies „The C Programming Language“.
Wichtige Besonderheiten für die C-Programmierung in Verbindung mit VxWorks sind:

- in Module müssen spezielle Headerfiles immer eingebunden sein (*vxWorks.h* und *taskLib.h*),
- die Startroutine muss nicht notwendigerweise *main* heißen,
- (Anwendungs-)Routinen sind direkt aus der Kommandozeile aufrufbar; Voraussetzung dazu ist, dass die Routinen programm-global, d. h. nicht mit dem Schlüsselwort `LOCAL` oder `static` deklariert sind,

Wie bei ANSI-C üblich, sind innerhalb eines Blockes keine Deklarationen von Variablen nach Anweisungen erlaubt. Der doppelte Slash ist kein gültiges Kommentarzeichen.

Achtung: Die Ausgabe von printf-Anweisungen bei Programmen mit nebenläufigen Prozessen erfolgt beim Start auf der Host-Shell nur teilweise und häufig nicht in der richtigen Reihenfolge. Ist die richtige Reihenfolge nötig, so ist ein Start aus der Targetshell nötig.

Programm *aufgabe0.c*

```
#include <pthread.h>
#include <stdio.h>

pthread_t thread1;
pthread_t thread2;

void* function1 (int*);
void* function2 (void*);

int main (void)
{
  printf("Start of main program\n");

  struct sched_param scheduling_parameters;
  pthread_attr_t thread_attributes;

  scheduling_parameters.sched_priority=sched_get_priority_max(SCHED_FIFO);

  if(0!=pthread_setschedparam(pthread_self(),SCHED_FIFO,&scheduling_parameters))
  {
    printf("failed to set scheduling parameters\n");
    return -1;
  }

  if(0!=pthread_attr_init(&thread_attributes))
  {
    printf("thread attribute initialization failed\n");
    return -1;
  }
  if(0!=pthread_attr_setinheritsched(&thread_attributes,PTHREAD_EXPLICIT_SCHED))
  {
    printf("failed to set scheduling attributes explicitly\n");
    return -1;
  }

  if(0!=pthread_attr_setschedpolicy(&thread_attributes, SCHED_FIFO))
  {
    printf("failed to set scheduling policy to fifo\n");
    return -1;
  }

  scheduling_parameters.sched_priority=260;

  if(0!=pthread_attr_setschedparam(&thread_attributes,&scheduling_parameters))
  {
    printf("failed to set scheduling parameters (priority)\n");
    return -1;
  }

  if(0!=pthread_create (&thread1, &thread_attributes, function1, NULL))
  {
    printf("could not create thread1\n");
    return -1;
  }

  scheduling_parameters.sched_priority=180;

  if(0!=pthread_attr_setschedparam(&thread_attributes,&scheduling_parameters))
  {
```

```
    printf("failed to set scheduling parameters (priority)\n");
    return -1;
  }

  if(0!=pthread_create (&thread2, &thread_attributes, function2, NULL))
  {
    printf("could not create thread2\n");
    return -1;
  }

  if(0!=pthread_join(thread1,NULL))
  {
    printf("error in joining thread1\n");
  }

  printf("End of main program!\n");
  return 0;
}

//function for thread 1
void* function1 (void* arg)
{
  string *str = ("Hello World!");
  print("Start of thread1!\n");
  for( i = 0; i < 10;)
  {
    printf("%s\n", str);
  }
  else
  {
    printf("Goodbye!\n");
  }
  if(0!=pthread_join(thread2,NULL))
  {
    printf("error in joining thread2\n");
  }
  printf("End of thread1\n");
}


/*function for thread2*/
void* function2 (void* arg)
{
  int x, y;
  printf("Start of thread2!\n");
  for( x =  0; x < 10; x++)
  {
    y /= sin(x);
    printf(".");
  }
  if(0!=pthread_join(thread1,NULL))
  {
    printf("error in joining thread1\n");
  }
  printf("\nEnd of thread2!\n");
}
```

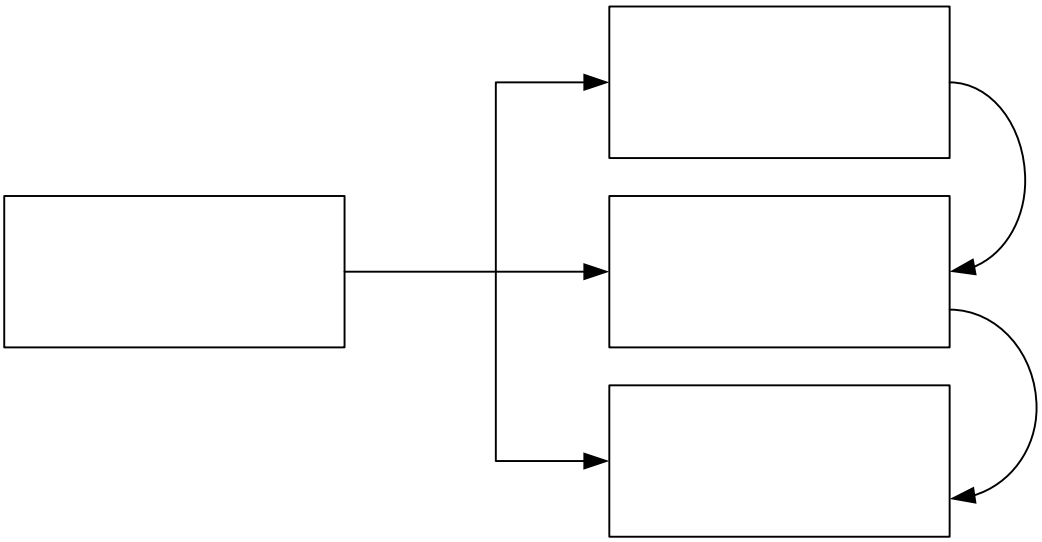
3 Aufgabe 1: Multitasking und Interprozesskommunikation

3.1.1 Ziel

In Aufgabe 0 haben Sie bereits Threads und Prioritäten kennengelernt. Zur Kommunikation zwischen einzelnen Threads wurden bisher globale Variablen (im Beispiel: thread IDs) benutzt.
Häufig ist es aber wichtig die Prozesse zu synchronisieren, d.h. Möglichkeiten zur Kommunikation zwischen Threads, die über die reine Benutzung von globalem Speicher hinausgehen, zu schaffen. Im Rahmen dieser Aufgabe soll deshalb die Verwendung von Semaphoren bzw. Nachrichtenwarteschlangen (Message Queue) erlernt werden.

3.1.2 Aufgabe

Schreiben Sie ein Programm, das drei parallel ablaufende Threads startet. Während der *erste Thread* umgehend mit der Ausführung beginnt, wartet der *zweite Thread* auf einen Semaphor, der von *Thread 1* freigegeben wird. *Thread 3* wartet solange, bis er per Message-Queue von *Thread 2* eine Nachricht erhält. Um den Programmablauf besser nachzuvollziehen zu können, ist es sinnvoll bestimmte Ausführungsetappen mit einer Meldung abzuschließen. Eine solche Meldung kann durch eine *printf*-Anweisung initiiert werden.



3.1.3 Implementierungshinweise

Die Benutzung von Semaphoren und Nachrichtenwarteschlangen wird in der POSIX-Anleitung, die sie auf unserer Homepage finden.

4 Aufgabe 2: zyklisch ablaufende Prozesse

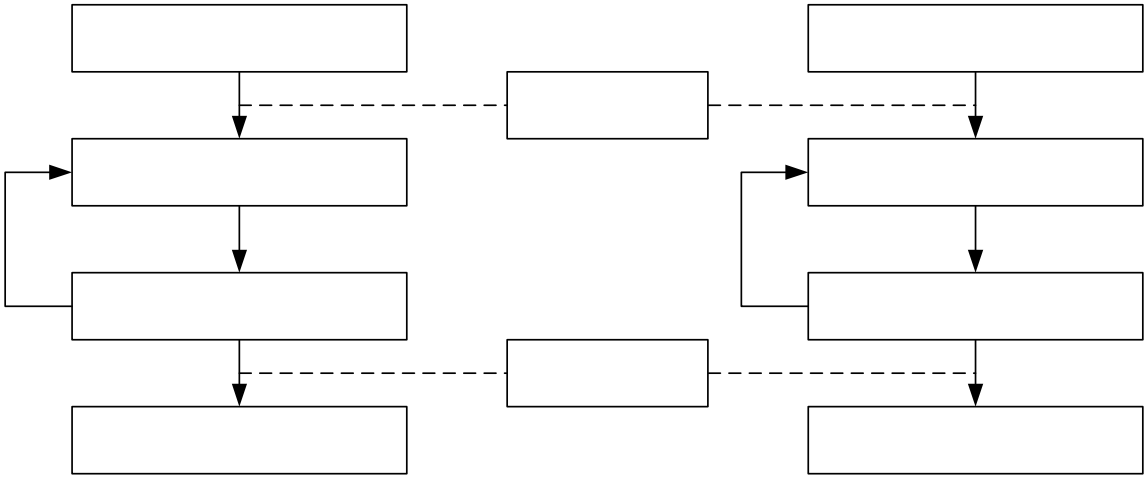
4.1.1 Ziel

Im Rahmen dieser Aufgabe sollen die Möglichkeiten Uhren zur Steuerung der Programmausführung zu benutzen erlernt werden. Dabei sollen sowohl die Möglichkeiten die Ausführung für ein bestimmtes Zeitintervall zu unterbrechen, als auch die periodische Stimulierung von Prozessen erprobt werden.

4.1.2 Aufgabe

Programmieren Sie je einen zyklisch aktivierten Prozess nach dem im Bild unten angegebenen Muster. *Prozess 1* soll nach der Prozessaktivität 100 Millisekunden warten und dann erneut starten. *Prozess 2* soll im Takt von einer 10 Hz (Periode 100 Millisekunden) neu gestartet werden. Geben Sie vor jeder Prozessaktivität die seit der *Startzeit* vergangene Zeit aus. Die *Endzeit* stellt einen Vergleichswert für den gesamten Zeitbedarf jedes Prozesses dar.

Zur Erläuterung der Unterschiede ist es nötig, dass die zur Ausführung der Prozessaktivität benötigte Zeit signifikant ist. Überlegen Sie sich eine geeignete Berechnung. Um ein unverfälschtes Ergebnis zu erhalten, müssen die Prozesse sequentiell ablaufen. Fügen Sie der Aufgabendokumentation eine Interpretation des unterschiedlichen Zeitverhaltens der beiden Prozesse bei.



4.1.3 Implementierungshinweise

Eine Beschreibung der Funktionen zur zeitlichen Steuerung von Prozessen finden Sie auf der Praktikumshomepage.

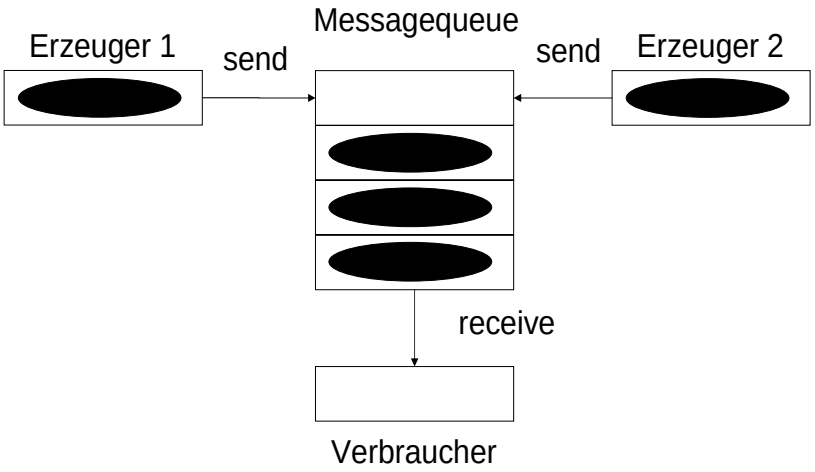
5 Aufgabe 3: Erzeuger-Verbraucher-Problem

5.1.1 Ziel

Ein typisches Problem in der Informatik sind Erzeuger-Verbraucher-Probleme, due in dieser Aufgabe umgesetzt werden sollen. Zur Kommunikation der Erzeuger und Verbraucher sollen Nachrichtenwarteschlangen eingesetzt werden.

5.1.2 Aufgabe

Starten Sie zwei Erzeugerprozesse mit gleicher Funktionalität. Jeder Prozess soll pro Sekunden je eine Nachricht in eine Message-Queue legen bis schließlich von jedem Prozess insgesamt 10 Nachrichten verschickt wurden. Das Fassungsvermögen der Queue sei auf 4 Einträge beschränkt. Starten Sie einen Verbraucher, der die Nachrichten alle 1,5 Sekunden abruft. Um den Verlauf nachvollziehen zu können, ist es sinnvoll, die diversen Schritte der einzelnen Prozesse auf der Konsole auszugeben.
Dokumentieren sie auch den Verlust einzelner Nachrichten aufgrund der Überfüllung der Nachrichtenwarteschlange.



Viel Spaß und Erfolg beim Lösen der Praktikumsaufgaben.