Versuch 1a: Einführung

Der Versuch 1 dient zum ersten Kennenlernen des OS9-Echtzeitbetriebssystems (OS9). Das **seriell angeschlossene Terminal (/term) soll im gesamten ersten Versuch** zur Bildschirm-Ausgabe der im Praktikum von Ihnen erstellten Programme genutzt werden.

V 1a.1 Erste Schritte zum Kennenlernen des OS9 Systems

Aufgabenstellung:

Stellen Sie vom PC aus eine weitere Terminalverbindung mittel **Telnet** her und loggen Sie sich nochmals als Benutzer super ein.
Probieren Sie folgende verschiedene Kommandos an beiden Konsolen aus:

```
pd,   pd -x,  dir -e /dd/SYS , mdir , procs -e, procs -a
```

- **Fragen zum *procs*-Ausdruck:**

Welche TaskIDs haben die Tasks die die Kommandos am Terminal und an der Telnet-Konsole von Ihnen entgegengenommen haben und welches Modul wird dabei verwendet? Welche Startpriorität haben diese? Wer startet procs?
Welche ist die Terminal- und welche die Telnet-Task? Wie sind die "Verwandschaftsbeziehungen"? Welches sind die active Tasks?
Welche Group/User-ID besitzen die Tasks?

- **Fragen zum *mdir*-Ausdruck:**

Welche Module sind geladen? Wie oft ist das Modul mshell geladen wie oft wird es verwendet?

- **Fragen zum *pd -x/ pd* -Ausdruck:**

Welche Verzeichnisse und in welcher Reihenfolge werden sie nach einem zu startenden Programm durchsucht? Welche Rolle spielt das Moduldirectory? Welche das Datendirectory?

V 1a.2. Editieren/Übersetzen/Testen von C-Programmen am OS9-System

Aufgabenstellung:

Es soll die Erstellung von einfachen C-Programmen im Zusammenspiel PC und OS9-System erlernt werden. Zur Erstellung wird am PC die HAWK-IDE (siehe Praktikumsplatzbeschreibung) verwendet.

Versuchsdurchführung Teil1:

Das zu erstellende C-Programm soll **hallo.c** heißen und wie folgt aussehen:

```
#include <stdio.h>
#include <const.h>
int main(int argc, char * argv[])
{
printf ("Hallo hier bin ich \n");
return 0;
}
```

Nach dem Erstellen und übersetzen dieser C-Source am PC sollte mittels **HAWK-IDE Load-Funktion** das Programm übertragen werden und am OS9-Terminal ausgeführt werden. Debuggen Sie das Programm von der HAWK-IDE aus.

Starten und testen Sie hallo!

Versuchsdurchführung Teil2:

Das Programm hallo soll im weiteren die Basis für Untersuchungen zur Veranschaulichung des unterschiedlichen Verhaltens von parallel ablaufenden Tasks unterschiedlicher Prioritäten sein.
Dazu soll die Task hallo verwendet werden, die mehrmals parallel gestartet werden soll. Damit man die einzelnen Aufrufe an ihrer Bildschirmausgabe unterscheiden kann soll nun die Task hallo ihre Task **id** , ihr Ausgabedevicenamen <**device-name**> und, falls vom Benutzer eingegeben, den ersten Aufrufparameter in einer Zeile **argv[1]** ausgeben.

Somit kann das Programm hallo wie folgt aufgerufen werden:
$ **hallo eins**

(➔ Wenn beim späteren Versuch 2: ***hallo*** zweimal gestartet wird -einmal mit Parameter und einmal ohne- kann somit unterschieden werden welcher Aufruf zuerst ausgeführt wurde).

Es soll nun folgender dreizeiliger Ausdruck erscheinen:

```
" eins  <TaskId> ---Zeile 1 –Geraet <devname> "
" eins  <TaskId> ---Zeile 2 –Geraet <devname> "
" eins  <TaskId> ---Zeile 3 –Geraet <devname> "
```

- **Hinweis: <TaskId>** soll die **aktuelle** Tasknummer sein, unter der das Programm läuft. Es gibt ein Betriebssystemaufruf ***getpid()*** zur Ermittelung der Tasknummer und den Aufruf ***_gs_devn()*** zu Ermittelung des <**devname**> von ***stdout=2***.
- **Jede Ausgabezeile soll durch einen seperaten *printf()*-Aufruf erzeugt werden**
  Achtung: Die Ausgabe ist gepuffert ➔ Eine Ausgabe am Bildschirm erfolgt erst bei Überlauf des Puffers **oder** durch Ausgabe eines "new-line" Zeichen ➔ in C ***"\n"***

Damit das Programm **hallo** sichtbar **Rechenzeit verbraucht** soll nach der **ersten** und nach der **zweiten** Ausgabezeile je **60 TIC** lang in einer Schleife (**aktiv!!)** gewartet wird. Schreiben Sie hierzu ein Unterprogramm Namens ***void waste(void)*** ,welches die untenstehende im Quelltext vorgegebene Funktion ***myclock()*** nutzt .

Ändern Sie nun das Programm hallo.c so, daß die Ausgabe der Task Id, des Devicenamens <device-name> und des ersten Aufrufparameters argv[argc-1] erfolgt und entsprechend die Rechenzeit verbraucht wird..

Testen Sie Ihr Programm. ➔ Aufruf hallo oder hallo <text> ➔ Machen Sie sich mit dem HAW-IDE-Debugger vertraut!

Funktion: `#include <types.h>, #include <sysglob.h>, #include <stddef.h>`

```
u_int32 myclock(void);
```

myclock() gibt die aktuelle Systemzeit in Anzahl der verstrichenen **tics** seit Start des Systems zurück.

```
u_int32 myclock(void)
{
glob_buff myg;
_os_getsys (offsetof(sysglobs, d_ticks ),sizeof(u_int32),&myg);
return myg.lng;
}
```

Versuch 1b: Starten und Beobachten von Tasks mit unterschiedlichen Prioritäten

Aufgabenstellung:

Es soll untersucht werden wie sich durch individuelle Prioritätensteuerung das Verhalten von Tasks beeinflussen läßt. Zu diesem Zwecke wird das fertige Programm **hallo** aus Versuch 1a.2 als Task mehrfach von der Terminal-Konsole aus gestartet und das Terminal als **gemeinsames** Ausgabemedium für alle gestarteten **hallo**-Tasks verwendet. Dadurch tritt eine mittels Prioritätensteuerung zu lösende Konkurrenzsituation um das Ausgabemedium Terminal auf.

Versuchsdurchführung:

Es sollen mehrere Fälle der Prioritätssteuerung untersucht werden. Je Fall werden immer zwei Tasks **hallo** mit unterschiedlichen Prioritätseigenschaften gestartet. Nach der Durchführung eines Falles notieren Sie sich bitte das beobachtete Ergebnis!!

- Für jeden Fall starten Sie zwei Tasks **hallo <parameter1/2> an der Terminalkonsole quasi gleichzeitig. Sie können durch <&> getrennt zwei Start-Kommandos in einer Zeile angeben.** Als letztes Kommando einer solchen Zeile verwenden Sie das <procs -e> Kommando, um sofort eine Übersicht über die im System laufenden Tasks zu erhalten.

- Die Ausgabe **beider** Tasks erfolgt auf /term, dem angeschlossenen Terminal

- Folgende beim jeweiligen Taskstart herbeizuführende Prioritätsfälle:

    1. Beide Tasks **gleiche** Priorität wie die mshell
    2. Beide Tasks **gleiche** Priorität z.B. **60,** aber deutlich **kleiner** als die der mshell (➔mshell prio=128)
    3. Beide Tasks **gleiche** Priorität **160** aber **größer** als die der mshell.
    4. Beide Tasks **ungleiche** Priorität , Task 1 **260/300** und Task 2 **300/260** somit beide **größer** als die der mshell

Achtung: 2 TICs ist ein TimeSlice. Jeder Prozess, dess Priorität kleiner 256 (maxage) ist, erhält die CPU im Minimum für die Dauer eines TimeSlices bevor wieder überprüft wird, ob ein Anderer höhere Priorität hat.

Fragen:

Protokollieren Sie bitte schriftlich die Antworten zu folgenden Fragen:

- *Was können Sie beobachten? Was passiert (Erklärung) am Terminal ?*
- *Was passiert nach Beendigung der gestarteten **hallo**-Tasks am Terminal?*
- *Diskutieren Sie die unterschiedlichen Ergebnisse der vier Fälle!*
- *Wie oft wird hallo geladen ?*

Versuch 2: Verarbeitung von Prozeß-Events im Taskkontext

Aufgabenstellung:

Es soll eine **Task** zur Reaktionsmessung entworfen werden. Dazu soll das LMS-HW-Board verwendet werden. Dort befinden sich zwei Taster und eine Reihe von acht LEDS. Die von Ihnen zu realisierende programmgesteuerte Reaktionsmessung soll wie folgt ablaufen:

1. Im Ruhezustand sind die LEDS aus
2. Das Drücken der rechten Taste startet den gewünschten Messablauf.
3. Nach Drücken der rechten Taste wird nach einer jedesmal unterschiedlichen Zeit von bis zu maximal 5 Sekunden die 8 LEDS angeschaltet. Das Anschalten der 8 LEDS ist die Aufforderung an den Benutzer so schnell wie möglich die linke Taste zu drücken
4. Die Reaktionszeit zwischen Angehen der 8 LEDS und dem Drücken der linken Taste durch den Benutzer wird millisekundengenau am Bildschirm ausgegeben; danach ist das Messprogramm wieder im Ruhezustand und wartet auf Start der nächsten Messung

Versuchsdurchführung (Hardware):

Das Experimentierboard ist an der sog. parallelen Schnittstelle des OS9-Systems angeschlossen. Zur Ansteuerung des Experimentierboards lesen Sie bitte in Kapitel 5 in OS920120709_tx_POS9-Praktpl.pdf nach.

Die Beobachtung der Funktion der Taster erfolgt einfach über das OS9-Kommando **events** . Jedes Drücken eines Tasters erzeugt eine Veränderung des betreffenden Eventwertes.

Versuchsdurchführung (Software):

Optional: Entwerfen Sie zunächst ein Zustands/Ereignisdiagramm für den Reaktionsmeßverlauf unter Verwendung der vorgegebenen Events zu den H1/H2 Ereignissen.

Pflicht: Implementieren Sie die Reaktionsmessung als eigenständige Task **remess** (C-Programm). Berücksichtigen Sie bitte folgende Randbedingungen zusätzlich:

1. automatische Beendigung nach drei Messungen
2. wird nach Start der Messung die linke Taste nicht gedrückt, so wird unendlich gewartet
3. wählen Sie die Taskpriorität so, daß andere Tasks Ihre Messung nicht beeinflussen können

Implementieren und Testen Sie die remess. Es ist die Abnahme der Task remess durch den Betreuer erforderlich!

Achtung alle Bibliotheksaufrufe geben einen Errorcode zurück: Werten Sie diesen aus: Abfrage
z.B. if ((ierr=_os_sleep(...)) != SUCCESS) exit (ierr); ➔ #include <const.h>

!!!! Achtung remess wird in Versuch 3 erweitert !!!

Includes: #include <types.h>, #include <sysglob.h>, #include <stddef.h>

Funktion: u_int32 myclock(void);

myclock() gibt die aktuelle Systemzeit in Anzahl der verstrichenen tics seit Start des Systems zurück.

```
u_int32 myclock(void)
{
glob_buff myg;
_os_getsys (offsetof(sysglobs, d_ticks ),sizeof(u_int32),&myg);
return myg.lng;
}
```

Versuch 3: Programmgesteuerte Tasksychronisation mittels OS9-Signale im Taskkontext

Aufgabenstellung:

Die Task **remess** aus Versuch 2 soll weiterentwickelt werden. Die Randbedingung des endlos Wartens nach Start der Messung (Drücken der rechten Taste) soll verhindert werden. Dazu wird bei Start einer Messung die Task **ueberwache** von **remess** aus gestartet. Erfolgt in **remess** die Betätigung der linken Taste innerhalb von 10 Sekunden, so schickt **remess** das Signal **310** an **ueberwache** und diese beendet sich sofort. Wenn die linke Taste nach 10 Sekunden nicht betätigt wurde, schickt **ueberwache** der Task **remess** das Signal **300 und** beendet sich sofort**.** Mit Erhalt des Signals **300** bricht **remess** das Warten auf das Event der linken Taste ab. Wird eine Messung auf diese Art abgebrochen, so gibt **remess** eine entsprechende Meldung ‚Timeout in Reaktionsmessung!' am Bildschirm aus und wartet wieder auf den Start einer neuen Messung durch das Drücken der rechten Taste.

Hinweis: Die Task **ueberwache** benötigt zur Signalübermittlung die ProzeßID (getpid()) von **remess**! Man kann der Task **ueberwache** in der Aufrufzeile (Stichwort argc/argv) jedesmal die ProzeßID von **remess** als Zahlenstring übergeben.

Versuchsdurchführung Teil1:

Die Task **ueberwache** soll zuerst erstellt und getestet werden. Dazu verschickt in dieser Version die Task **ueberwache** kein Signal 300, sondern gibt stattdessen einen Text am Bildschirm aus. **ueberwache** reagiert somit auf das Signal 310 und verarbeitet einen Zahlenstring als 1. Aufrufparameter, den sie zur Kontrolle ebenfalls am Bildschirm ausgeben. Die Task **remess** wird zunächst simuliert durch geeignete Verwendung des mshell-Kommandos *signal*.

Hinweis: Sie können mittels des Kommandos ***signal <process-id> <signalwert>*** auf **mshell-Kommandoebene** ein Signal an die Task <*process-id*> verschicken.

- **ueberwache** wird manuell als Task **parallel zur mshell** an der Terminal-Konsole gestartet
- **Überwache** soll nicht unnötig CPU-Zeit verbrauchen.

Gehen Sie schrittweise vor!!

Für die programmgesteuerte Signalbearbeitung stehen mehrere Betriebssystemaufrufe (C-Library) zur Verfügung:

- ***_os_intercept(), _os_sleep()*** *,* ***_os_send(), _os_rte().***

Abnahme durch den Betreuer!

Versuchsdurchführung Teil2:

Erweitern und testen Sie nun die Task **remess und modifizieren Sie ueberwache** so, daß die Aufgabenstellung erfüllt ist.

Abnahme durch den Betreuer!

Versuch 4: Taskkommunikation mittels "shared- memory"

Aufgabenstellung 4a:

Dieser Versuch baut auf den Ergebnissen des Versuchs 3 auf. Die Übergabe der ProzeßID von **remess** an **ueberwache** soll nun mittels shared memory (OS9-Datenmodul) erfolgen. Ansonsten soll sich am Ablauf wie in Versuch 3 Teil 2 beschrieben nichts ändern. Der Name des von Ihnen anzulegenden Datenmoduls sei **datmod** und es sei folgende Struktur von Ihnen im Datenmodul abgelegt:

```
struct Datmod {
      u_int32 remess_id;
      u_int32 ueberwache_id;
      u_int32 steuer_id;
      float  rezeit;    /* erzielte Reaktionszeit in Sekunden */
      u_int32 timout;   /* Timeoutzeit in sec bis Versuch abgebrochen wird */
            };
```

Das Datenmodul wird von **remess** angelegt. **remess** speichert seine ProzeßID in remess_id und die erzielte Reaktionszeit nach jeder Messung in rezeit ab. **ueberwache** linkt sich auf das Datenmodul und kann dann die remess_id und den timout lesen und verwenden. Außerdem legt **remess** die ProcessID von **ueberwache** in ueberwache_id ab. Bei Beendigung von **remess** ist **remess** für die Entfernung des Datenmoduls aus dem Speicher verantwortlich.

Versuchsdurchführung 4a:

Erweitern Sie **remess** und **ueberwache** entsprechend den Vorgaben. **Abnahme durch den Betreuer!**

Aufgabenstellung 4b:

Nun sollen zusätzliche Benutzerinteraktionen am Terminal durch die Task **steuer** erfolgen. Die Task **remess** beendet sich nicht mehr nach drei Durchläufen automatisch.. Die Task **remess** startet zu Beginn einmal die Task **steuer** und legt deren ProzeßID in steuer_id ab.

Die zu entwickelnden Tasks **steuer** , **ueberwache** und **remess** erfüllen folgende Aufgaben:

1. **steuer** wartet auf Benutzerkommandos vom Terminal. Mögliche Kommandos und Wirkungen:
   - b ➔ Beendigung von remess, ueberwache und steuer. remess räumt alles auf
   - l ➔ Ausgabe der zuletzt gemessenen Reaktionszeit in Sekunden bei einer Genauigkeit von 10 ms
   - p ➔ Ausgabe der eingestellten Timeoutzeit
   - t <n> ➔ neue Timeoutzeit in n Sekunden für ueberwache in Variable timout hinterlegen
2. **ueberwache:** wertet die Variable timout für Einstellung der maximal erlaubten Dauer bis zum erzwungenen Abbruch aus
3. **remess:** Ausgabe der Reaktionszeit nach jeder Messung, sonst Funktion wie in Aufgabe 4a; **Sie** beendigt sich als letzte Task im Fall des b-Kommandos und entfernt alle angelegten events und das Datenmodul. **remess** legt nach jeder Messung in der Variablen rezeit die erzielte Reaktionszeit ab. Ist die Zeit groesserr Timeout, dann gibt **remess** eine Warnung aus.

Achtung: steuer darf nur Zeichen von stdin lesen wenn sichergestellt ist, dass welche da sind. Ein Lesen ohne dass Zeichen da sind blockiert jedwede Ausgabe –egal welche Task- an das Terminal ausgibt! Mittels der Funktion *_os_gs_ready()* läßt sich überprüfen ob ein Zeichen da ist (path=stdin=0).

Zur Erweiterung der bisherigen Synchronisation von **steuer, ueberwache** und **remess** verwenden Sie bitte ausschließlich **signale** oder **events.**

Versuchsdurchführung:

Erstellen Sie **steuer** und erweitern Sie **remess** und **ueberwache** entsprechend den Vorgaben.
Abnahme durch den Betreuer!

Versuch 5: Threadprogrammierung

Aufgabenstellung Reaktionszeitmessung All-in-One:

Die Reaktionszeitmessung soll nun nur noch durch eine Task **reakt** erledigt werden. Die Funktionalität die **ueberwache** und **steuer** bereitstellen, sind nun mittels **Threads innerhalb Task reakt** zu realisieren.

Zusammengefasst: **Ausgehend** von Versuch 4b sollen nun alle Aufgaben **remess, steuer und ueberwache** durch Threads in **einer Task reakt** erledigt werden.

Im Einzelnen:

- Das main von **reakt** ist im wesentlichen **remess.c** und anstelle des Startens von **ueberwache** bei jeder Messung wird nun ein Thread gestartet der die Funktion **ueberwache** ausführt (Die Funktion ueberwache ist hierbei das von Ihnen modifizierte main der Task ueberwache).
- Ebenso wird am Anfang der Task **reakt** ein Thread gestartet der die Funktion **steuer** ausführt.
- **Die Verwendung der Events bleibt unverändert**.
- Aber es gibt nur **eine** Signalinterceptfunktion in **reakt.c**. Damit kann nur der **Hauptthread von reakt** Signale empfangen. Die Funktion **ueberwache** und **steuer** können demzufolge nur Signale versenden aber nicht mehr empfangen. Deshalb müssen diese geeignet umgeändert werden. Threads wie **ueberwache** können auch mittels `pthread_cancel()` vom **Hauptthread** der Task **reakt** aus beeendet werden.

Vorbereitung (1) :

- Legen Sie eine neue Komponennte in der HAWK-IDE und legen Sie die (leere) Datei **reakt.c** an; fügen Sie sie zum Projekt hinzu. Danach kopieren Sie im Editor mittels cut&paste die Inhalte der Dateien **ueberwache.c, steuer.c und remess.c** in Ihre Datei **reakt.c**.
- Aus dem main des kopierten Inhalts von ueberwache.c in reakt.c machen Sie ein **void ueberwache() ➔ueberwache Threadfunktion**
- Aus dem main des kopierten Inhalts von steuer.c in reakt.c machen Sie ein **void steuer() ➔steuer Threadfunktion**
- Das main() von **remess** beibt das main des Programms **reakt.c**
- Löschen Sie alle **#include <..>** Anweisungen, die nicht am Anfang Ihrer Datei stehen
- Ersetzten Sie die **_*os*_exec()**-Aufrufe durch **entsprechende Threaderzeugeaufrufe**

Anstelle des Datemmoduls **datmod** kann! nun eine taskglobale Variable des gleichen Typs **struct Datmod** eingesetzt werden. Da die Threads alle unkoordiniert auf die gemeinsame globale Variable zugreifen können ist **in allen Fällen** (also: bei Datenmodul oder globaler Variable) eine **Zugriffs-Synchronisation notwendig**. Dazu verwenden Sie **eine** von Ihnen global angelegte **pthread_mutex-**Variable und serialisieren jeden Thread-Zugriff auf die gemeinsame globale Variable über pthread_mutex_lock() /pthread_mutex_unlock()-Aufrufe.

Vorbereitung (2):

- Kann: Entfernen Sie alle Aufrufe, die das Datenmodul **datmod** betreffen
  - Legen Sie eine globale Variable names **gdat** und vom Typ **struct Datmod an.**
  - Ersetzen Sie alle alten Zugriffe **myzeig->** über den Zeiger in Datenmodul durch **gdat.**
- Schreiben Sie eine Funktion **void lock(void); und void unlock(void)** die die pthread_mutex_lock() /pthread_mutex_unlock()-Aufrufe kapselt.
- Ummanteln Sie jeden Zugriff auf **gdat.**xxx/ **myzeig->** mit **lock(); gdat.**xxx**;/ myzeig-> ; unlock();** um innerhalb der Umklammerung alleinigen Zugriff auf **gdat.** xxx zu erhalten.

Bringen Sie Ihr reakt zum laufen. Abnahme durch den Betreuer!

Versuch 6: Taskkommunikation mittels "named-pipes"

Aufgabenstellung:

Es sollen schrittweise zwei Tasks, eine **Sendertask: sendm** und eine **Empfängertask: empfm,** von Ihnen entwickelt werden, die synchronisiert Nachrichten austauschen.

Das Versenden von Nachrichten von Task zu Task soll zunächst mittels OS9 **"named-pipes"** erfolgen. Dazu wird vereinbart, daß die named-Pipe den Namen **"/pipe/mess_buf"** erhalten soll und **64 Bytes** groß sein soll.

Es stehen Ihnen die OS9-Aufrufe *_os_create(), _os_open(), _os_read(), _os_write()* und *_os_close()* zur Verfügung.

Bei jeder Nachrichtenübermittelung sollen immer eine Nachricht gleich 64 Bytes verschickt werden; egal wie lang der eingegebene Text ist! Der \0 Character schließt den druckbaren Text ab.

Vorgaben zur **Sendertask**:

Die Sendertask nimmt am **Telnet**-Terminal eine beliebige Textnachricht, die der Benutzer frei an der Tastatur eingeben kann, entgegen und versendet sie an den Empfänger.

Die Textnachricht darf 63 Zeichen (64. Zeichen =Endezeichen \0) **nicht** überschreiten. Die Benutzereingabe wird mit der Eingabetaste abgeschlossen. Wird kein Text eingegeben. sondern nur die Eingabetaste betätigt, so schickt die Sendertask die '**Leer'-Nachricht (\0)** an den Empfänger und **beendet sich danach**.

Die Sendertask darf erst wieder eine Nachricht vom Benutzter entgegennehmen und an die Empfängertask schicken, wenn diese die vorherige Nachricht bereits abgeholt hat.

Vorgaben zur **Empfängertask**:

Die Empfängertask hat als Ein/Ausgabegerät zum Benutzer die **Terminal-Konsole**. Sobald für die Empfängertask eine Nachricht vorliegt, soll zunächst, **ohne die Nachricht schon abzuholen,** am Terminal die folgende Meldung an den Benutzer erfolgen:

"Nachricht da! Zum Anzeigen drücken Sie eine beliebige Taste"

Erst **nach der Eingabe eines beliebigen Tastendrucks** am **Telnet**-Terminal **holt** die **Empfängertask die vorliegende Nachricht** ab und gibt sie am Terminal aus.
Gleichzeitig sorgt die Empfängertask dafür, daß gegebenenfalls der Sender eine neue Nachricht überstellen darf.
Erhält der Empfänger eine **Leer-Nachricht** so beendet sich die Empfängertask automatisch, ohne Rückkehr zu einem Eingabeprompt, nach Ausgabe der Meldung:

"Nachrichtenübermittlung eingestellt!"

Die Sendetask wird manuell gestartet. Diese startet nach dem Anlegen der Pipe die Empfangstask.
Achtung: Die laufende *mshell* am Terminal */term* muß mittels *sleep 0* schlafen gelegt werden, damit eine Eingabe an das *empfm*-Empfangsprogramm möglich wird (VOR START des *sendm*-Programms)

Versuchsdurchführung Teil1:

Im ersten Schritt soll aktiv im Empfänger mittels Polling auf den Erhalt einer Nachricht gewartet werden.

Skizzieren Sie vor Beginn der Programmierung graphisch die Lösung! Vor dem Beginn der Programmierung ist eine Abnahme der Lösungskizze erforderlich!

Die Skizze muß folgende Elemente enthalten:

- *Getrennt Skizzenteile für Task* ***sendm*** *und* ***empfm***
- *Initialisierungszustand pro Task*
- *Endezustand pro Tasks*
- *Ein/Ausgabe - Zustände*

- ***Programmieren und Testen Sie die skizzierte Lösung***

Versuchsdurchführung Teil2:

Jetzt soll der Empfänger auf den Erhalt einer Nachricht ohne CPU-Zeitverbrauch warten. Dazu gibt es bei Pipes die Möglichkeit sich ein Signal schicken zu lassen, wenn Daten in die **voher leere** Pipe geschrieben werden. Die Signalnummer ist frei vorgebbar; wir verwenden die Signalnummer **2000**. Mittels *_os_ss_sendsig()* teilt man der named Pipe mit, daß man ein Signal geschickt haben möchte, wenn Daten ankommen (oder mittlerweile da sind). Erhält man das Signal so **muß** mittels *_os_ss_relea()* die Signalvereinbarung vor der nächsten Benutzung wieder aufgehoben werden. Wartet bereits ein Anderer auf Daten der Pipe mittels des gleichen Mechanismusses, so erhält man vom *_os_ss_sendsig()* Aufruf eine entsprechende Fehlermeldung und die Signalanforderung schlägt fehl.
Mittels *_os_sleep()* kann man auf das Auftreten des Signals warten ➔Außerdem setzt diese Aufruf automatisch *sigmask* auf 0 und erlaubt somit den Empfang von Signalen auch wenn vorher *sig,mask* auf 1 war.

Hinweis: Mittels *_os_sigmask(1) -gesperrter-Code-_os_sigmask(0)/_os_sleep() lassen sich Programm-Abschnitte gegen den Empfang von Signalen sperren*

- ***Programmieren und Testen Sie Teil 2***

Versuch 7 Parkhaussimulation

Ziel des Versuchs ist die Erlernung des Umgangs mit vorgegebener Prozeßperipherie und Treibersoftware.

Es steht das LMS-HW-Board zur Verfügung. Dieses ist an die parallele Schnittstelle des OS9-Systems angeschlossen. Die Benutzungsanleitung des LMS-HW-Boards ist ausführlich im Umdurck "OS9-Praktikumsplatzbeschreibung" zu finden.

V 7.0 Entwickeln einer Testapplikation zur Ansteuerung des Experimentierboards

Es sollen eine Testapplikation, bestehend aus drei Tasks **testio, H1int und H2int,** entwickelt werden:

1. Die Task **testio** soll permanent die Schalterstellungen des Experimentierboards abfragen und bei **Veränderung** deren Stellung durch die LEDs anzeigen lassen.
2. Drückt man nun die linke Taste H1 so soll von der Task **H1int** an der Konsole die Nachricht "Taste H1 gedrueckt" ausgegeben werden.
3. Drückt man nun die rechte Taste H3 so soll von der Task **H2int** an der Konsole die Nachricht "Taste H2 gedrueckt" ausgegeben werden.
4. **testio** soll sich und die anderen Tasks **H1int** und **H2int** beenden wenn die Schalterstellung **$AA** eingestellt ist.
5. **testio** startet zu Beginn die Tasks **H2int und H1int.**

Verwenden Sie zum Starten von Tasks hierzu den **_os_exec() Aufruf mit Subfunktion _os_fork** und zum Warten auf Beendigung der Kinder den **_os_wait()** Aufruf. Kapseln Sie den **_os_fork()**-Aufruf in ein wiederverwendbares Unterprogramm my_fork(), der als Schnittstellenparameter die für diese Aufgabe notwendigsten Parameter: Taskname, 1.Argument der Aufrufzeile, TaskId, Priorität bereitstellt und den Rest "konstant" vorbelegt.

Versuchsdurchführung:

- ***Programmieren und testen Sie Ihre Lösung.***

V 7.1 Parkhaussimulation

Für diesen Versuch wird das Parkhausbeispiel aus Kapitel 1 der Vorlesung als bekannt vorausgesetzt.
Es soll in diesem Versuch die Parkhaussteuerung programmiert werden. Zur Simulation der Ein- und Ausgangsgrößen wird das Experimentierboard verwendet.

Das Parkhaus besitzt 15 Stellplätze. Der Einfahrtkontakt wird durch die linke Taste H1 simuliert. Jeder Tastendruck stellt ein einfahrendes Auto dar. Der Ausfahrtkontakt wird durch die rechte Taste H2 simuliert. Jeder Tastendruck stellt ein ausfahrendes Auto dar . Die Ausfahrtschranke (gedacht) öffnet automatisch bei Betätigung des Ausfahrtkontaktes. Die Einfahrtschranke (gedacht) schließt automatisch wenn die Anzeige "besetzt" zeigt und somit keine Fahrzeuge mehr einfahren dürfen. Wird dann der Einfahrtkontakt betätigt, so soll dies vom Programm ignoriert werden. Sind keine Autos mehr im Parkhaus soll weiteres Betätigen des Ausfahrtkontaktes eine Warnmeldung am Terminal erzeugen, daß der rechnergeführte Fahrzeugbestand im Parkhaus nicht stimmt. Die Anzahl der im Parkhaus befindlichen Fahrzeuge sollen mittels 4 LEDs (LEDs3-0) binärcodiert am Experimentierboard angeezeigt werden. Die 4 verbleibenden LEDs dienen zur Simulation der Anzeige "belegt"/"frei". "frei" entspricht alle vier LEDS aus und "besetzt" entspricht alle vier LEDs an.Zu Beginn ist das Parkhaus leer. Die Schalter am Experimentierboard dienen zur Bedienung der Parkhaussteuersoftware.
Grundstellung sei alle Schalter "aus" entspricht Datum $00. Schalter 7 "aus" bedeutet Normalbetrieb. Schalter 7 "ein" bedeutet Anzeige auf "besetzt" . Schalter 6 "ein" bedeutet übernehme die Schalterkombination Schalter 3-0 als neuen Fahrzeugbestand im Parkhaus. Schalter 5 "ein" bedeutet beenden der Parkhaussteuersoftware.

Skizze einer möglichen Lösung:

Die Parkhaussteuerung besteht aus vier Tasks **Anzeige, Input, Einfahrt und Ausfahrt,** die über ein Datenmodul **"Parkhaus"** miteinander kommunizieren. **Input** startet alle Tasks und beendet sie ggf. auch wieder.

Datenmodul "Parkhaus":➔ Enthält alle notwendigen ProcessIDs, Fahrzeugbestand, Sperrung der Einfahrt
➔Zugriff über Semaphore!!

Input: ➔startet alle anderen Tasks **und** beendet sie ggf. ➔legt das Datenmodul an ➔kreiert die notwendigen Events ➔wertet die Schalterstellungen aus ➔erlaubt Fahrzeugbestandsänderung ➔ Sperrung der Einfahrt veranlaßt "besetzt" wenn Schalter 7 ein

Anzeige:➔Zeigt alle LEDs an, ➔Triggerung, daß was Neues angezeigt werden muß ➔ Realisierung der Triggerung mittels **Event** oder **Signal**

Einfahrt:➔Verarbeitet das Einfahrsignal ➔ veranlaßt Anzeige "besetzt"➔erhöht ggf. Bestand

Ausfahrt:➔Verarbeitet das Ausfahrsignal ➔ veranlaßt Anzeige "frei" ➔ erniedrigt den Bestand

- ***Entwerfen, Realisieren und Testen Sie die Parkhaussteuerung am OS9-System***