Programmierung

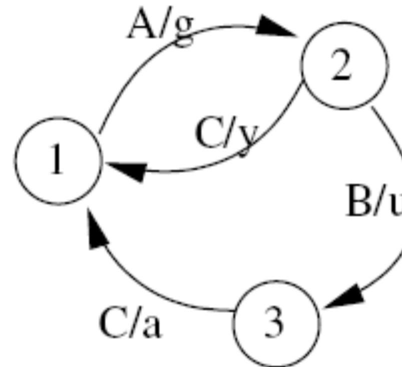
Programmier-Methoden bei Echtzeitsystemen

- **SPS-Programmierung (EN 61131 / EN 61499)**
  - Anweisungsliste (AWL) (Text)
  - Strukturierter Text (ST) (Text)
  - **Ablaufsprache (AS) (Text,Graphisch)**
  - Kontaktplan (KOP) (Graphisch)
  - **Funktionsbausteinsprache (FBS)**
- **Zustandsautomaten (state machines)**
- **Programmiersprachen (typisch C/C++)**

Programmierung

Zustandsautomaten (finite state machine)

- **Weit verbreitetes Paradigma bei Ereignis-getriebenen Systemen (z.B. Benutzerinteraktion Mobiltelefon)**
- **Zustandsautomaten bestehen aus einer Anzahl Zuständen, Ereignissen, Transitionen und Ausgaben**
- **System befindet sich in genau einem Zustand**
- **Ereignis führt zu Zustandsübergang in einen Folgezustand**
- **Implementierbar durch**
  - **(allgemeinen) Interpreter**
  - **Beschreibung (Automatentafel)**



	A	B	C
1	2/g	–	–
2	–	3/u	1/y
3	–	–	1/a

Programmierung

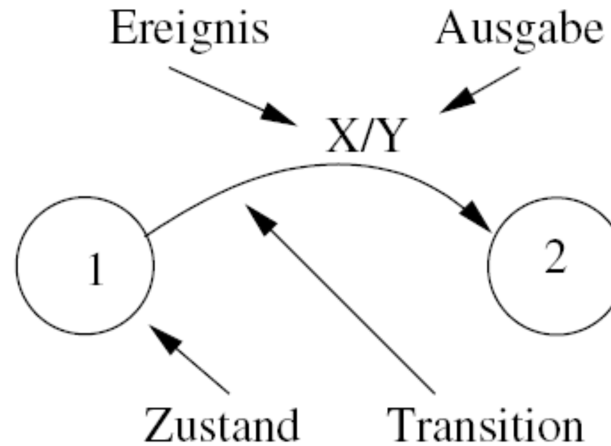
Zustandsautomaten (finite state machine)

Zustandsautomaten mit Ausgaben

Mealy-Automat:

- **Ausgabe mit Transition assoziiert**

(z.B. Mobiltelefon)

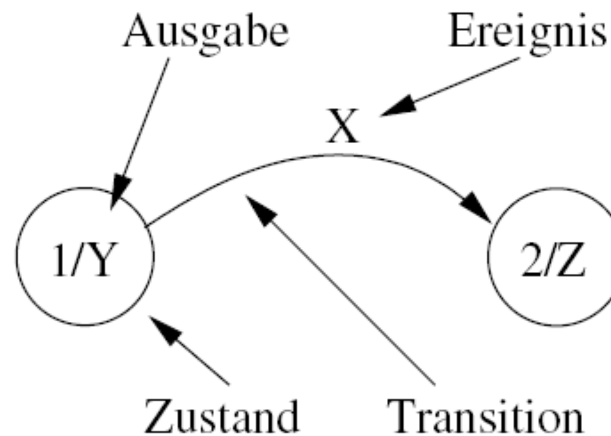


Moore-Automat:

- **Ausgabe mit Zustand assoziiert**

(z.B. Ampel)

Ineinander überführbar



Programmierung

Zustandsautomaten (finite state machine)

Kennzeichen:

- **In HW oder durch ein einfaches Laufzeitsystem realisierbar**
- **Geeignet für digitale Ein-/Ausgänge (Ereignisse, diskrete Zustände)**
- **Nicht geeignet für analoge Prozessdaten (kontinuierliche Zustände)**
- **Übersichtlich**
- **Ohne dedizierte Programmierkenntnisse programmierbar**
- **Sicher, da eine Überprüfung sowohl des Abdeckungsgrades als auch der Laufzeit (deterministisch) möglich ist.**

Programmierung

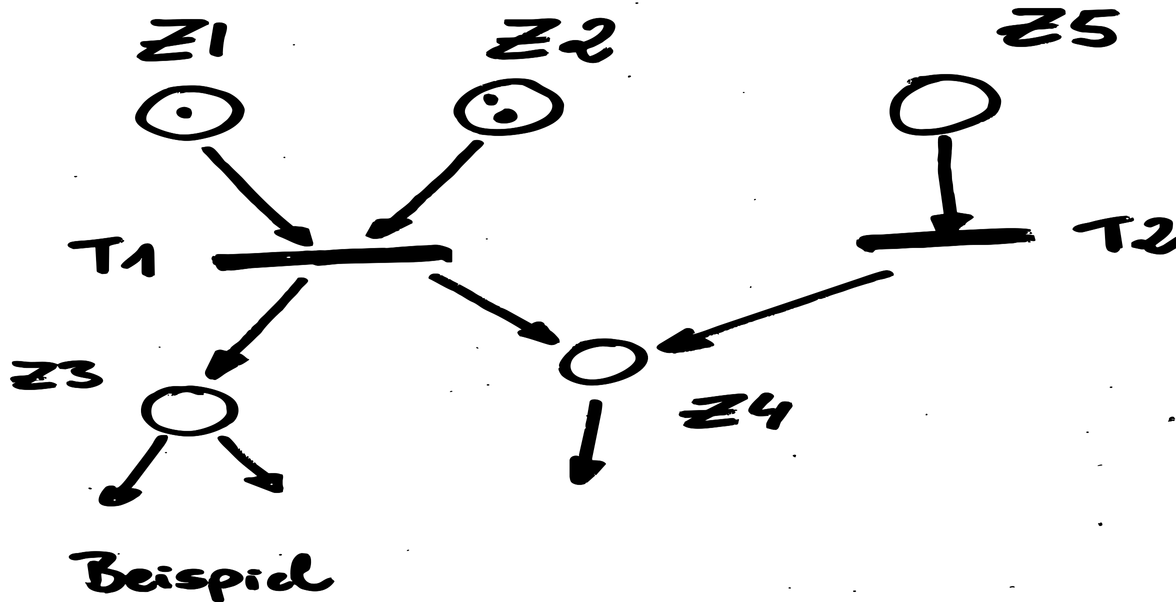
Petri-Netze I („Verallgemeinerung Zustandsautomaten“)

- Petrinetze sind ein Hilfsmittel zu Darstellung nebenläufiger Prozesse mittels gerichteter Graphen
- Petrinetze bieten die Möglichkeit übersichtlich Synchronisationsaufgaben zwischen beteiligten Prozessen zu veranschaulichen und festzulegen. Es gibt:
- MARKEN, STELLEN und TRANSITIONEN
- STELLEN und TRANSITIONEN haben Ein- und Ausgänge
- STELLEN und TRANSITIONEN werden über unidirektionale gerichtete Pfeile miteinender verbunden
- STELLEN und TRANSITIONEN wechseln sich ab; es werden NIE zwei TRANSITIONEN und/oder STELLEN aufeinenderfolgend verbunden.
- Die STELLEN entsprechen den Zuständen, die ein Prozeß durchlaufen kann
- aktive STELLEN enthalten MARKEN, inaktive keine
- jede STELLE besitzt eine MAXIMALE Aufnahmekapazität von MARKEN, default: unendlich viele Marken möglich
- TRANSITIONEN stellen (Weiterschalt-)Bedingungen, um den Prozeßzustand von STELLE zu STELLE weiterzuschalten.
- ➔ Diese Bedingungen können als boolsche Gleichungen formuliert sein.
- ➔ Die Bedingung verknüpft alle über die gerichteten Pfeile angeschlossenen STELLEN
- ➔ TRANSITION nur aktiv wenn Bedingungen erfüllt

Programmierung

Petri-Netze II („Verallgemeinerung Zustandsautomaten“)

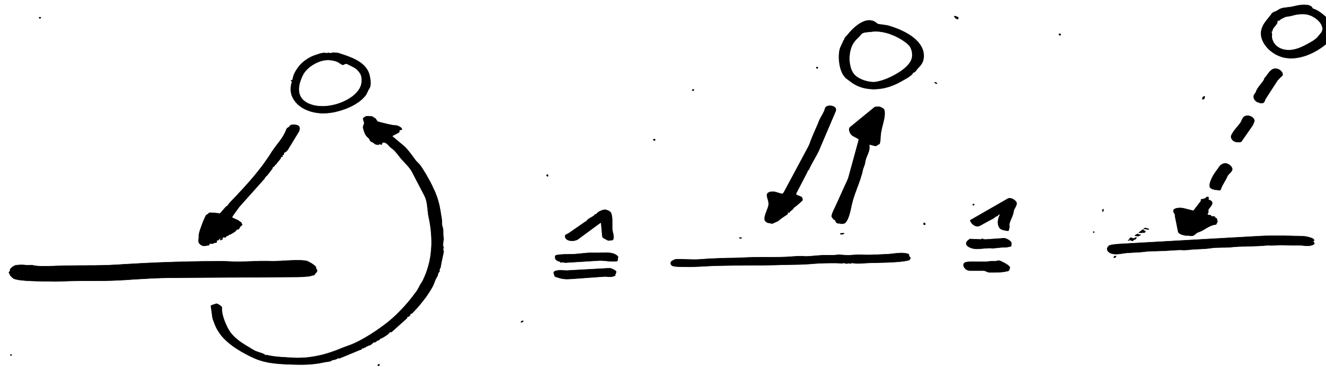
- **SCHALTREGEL:**
- **➔Jede TRANSITION schaltet automatisch, wenn jede einzelnen STELLE, die mit dem Eingang der TRANSITION verbunden ist, mindestens EINE MARKE enthält UND die TRANSITIONS-Bedingung erfüllt ist.**
- **➔ Im Schaltvorgang wird jeder einzelnen STELLE, die am Eingang angeschlossen ist, EINE MARKE abgenommen**
- **➔ Jede am Ausgang der Transition angeschlossenen STELLE erhält genau EINE MARKE (es gibt unendlich MARKEN!!)**
- **➔ STELLEN die vorher keine MARKE hatten werden dann aktiv!**
  ➔STELLEN die dabei alle MARKEN verlieren werden inaktiv!
  ➔ (theoretisch) alle Transitionen schalten ‚gleichzeitig nebenläufig'
- **Beispiel:**



Programmierung

Petri-Netze III („Verallgemeinerung Zustandsautomaten“)

- **Abkürzungen:**



- **Petrinetze lassen sich leicht formal auf Konsistenz überprüfen**

➔ Prüfung ob alle STELLEN je MARKEN erhalten (also aktiv sind)

➔Verklemmungsdetektion: gehen Marken verloren, oder finden keine TRANSITIONEN statt

➔Kapazitätsüberschreitung: Marken werden produziert (➔ siehe Ausgang TRANSITIONEN) und nicht verbraucht; es wird die STELLEN-Kapazität überschritten

Programmierung

Petri-Netze IV („Verallgemeinerung Zustandsautomaten“)

- **Beispiel 2:**

Nachrichtenstelle:

Task1

Task2

S1

V1

T1

S2

Trigger

N

T2

Nachricht ..... wird verarbeitet

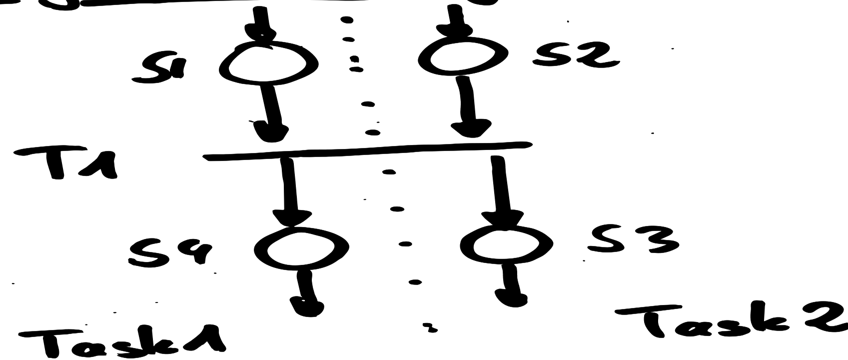
≙ Trigger entleert

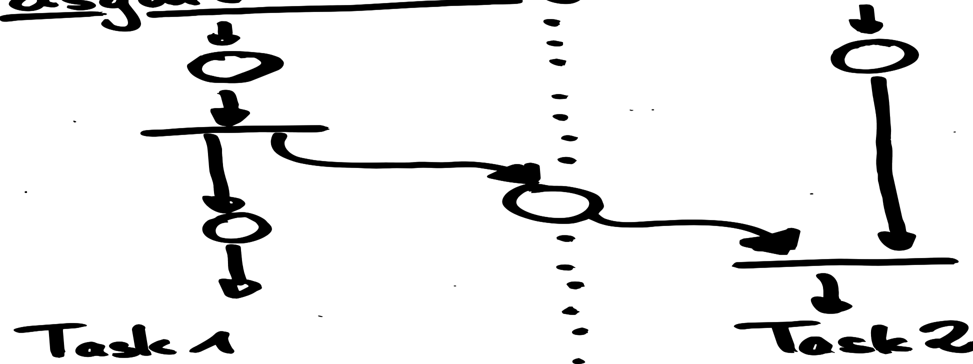
T3

Programmierung

Petri-Netze V („Verallgemeinerung Zustandsautomaten“)

- **Beispiel 3:**





Programmierung

Programmiersprachen

Assembler

Selten, bei sehr zeitkritischen Anwendungen (Treiber, hardwarenah)

C

Die Programmiersprache für Realzeitsysteme

- Effiziente Codeerzeugung
- Weite Verbreitung
- Hardwarenahes Programmieren ist möglich
- Input-/Output-Routinen sind *nicht* in der Sprache verankert
- Compiler für beinahe alle CPUs
- Programmiersprache ist unabhängig vom Betriebs- bzw. Laufzeitsystem.

C++

- Weniger häufig als C
- Hochoptimierende Compiler
- Tiefes Wissen über Effizienz erforderlich (zB. Matrizen: A = B * C;)
- Versteckte Speicherallokation

Programmierung

Programmiersprachen

(Safe-)PEARL nach DIN 66253

- PEARL (Process and Experiment Automation Realtime Language)
- Realzeit-Programmiersprache, eng mit Pascal verwandt
- Realzeit-Konstrukte als Bestandteil der Sprache (Synchronisation durch Semaphore; Beeinflussung des Scheduling)

ADA ISO/IEC 8652:*2012*

- Schöne, sichere Sprache
- Wenig verbreitet (Luft-und Raumfahrt, Militär, vom DOD gefördert)
- Threading- und Synchronisationskonzepte
- Saubere Modularisierung
- Schwer hardwarenah zu programmieren
- Validierte Compiler!!!

Programmierung

Programmiersprachen

JAVA -> RealTime Specification JAVA RTSJ 2019

Zunehmende Bedeutung

- als virtual machine, unter einem anderen (Realzeit-) Betriebssystem
- als Laufzeitsystem selbst

Prinzipiell gute Eignung

- Objekt-Orientiertheit
- eingebautes Thread-Konzept
- Ausnahme-Behandlung
- weniger anfällig für Programmierfehler als C oder C++
- sehr gute Portabilität
- weit verbreitetes Know-How
- hohe Programmier-Produktivität

ABER:

Java classisch nicht deterministisch (Memory-Management: Garbage-Collection; RT-Java: unterschiedliche Speicherklassen)

FAZIT: Geeignet für Soft-Realtime-Systeme

Programmierung

Programmiertechnik bei Echtzeitsystemen

Voraussetzungen

- Zerlegung der Aufgabe auf mehrere (konkurrierende und kooperierende) Tasks, um schritthaltende Verarbeitung zu garantieren (Multitasking)
- Unterbrechbarkeit (Preemption) der Tasks

Folgerungen

- Kontrollfluss zwischen den einzelnen Tasks
  - Synchronisation, z.B. durch Semaphore und Events
- Datenaustausch zwischen Tasks
  - Inter-Prozess-Kommunikation (IPC)
- Scheduling, Priorisierung

Programmierung

Kontrollfluss

Beispiel: Pathfinder Mars-Mission (Landung: 4. Juli 1997)

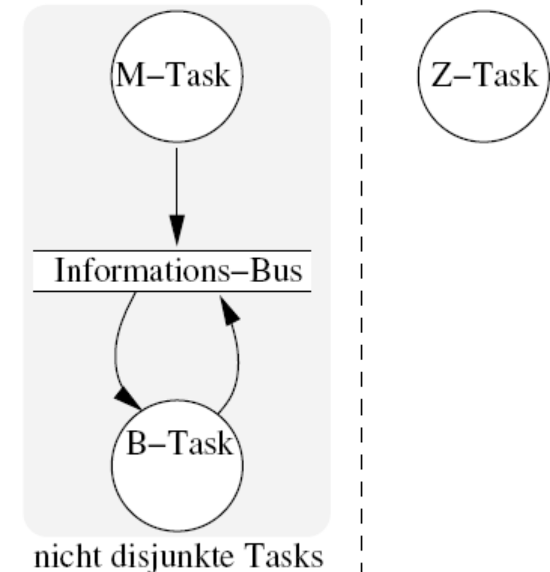
- Lander und Rover (Sojourner)
- Steuerungssoftware unter VxWorks
- Prinzipiell: Erfolgreiche Landung, alle Systeme nominell
- Aber: Unerklärliche Systemresets
- Wurde zum Klassiker/Lehrstück

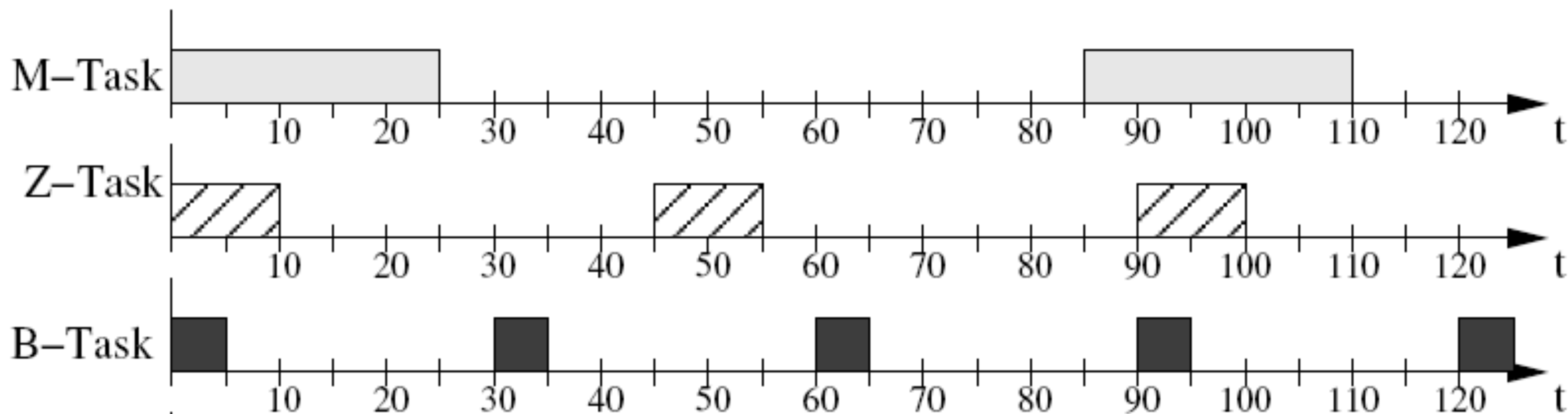
Programmierung

Beispiel: Pathfinder-Mars-Mission

Prinzipielle Systemstruktur

- **Z-Task**
  - **Housekeeping (periodische Zustandsüberwachung, z.B. Energiereserven, Temperaturen, Ströme, Odometrie,..)**
- **M-Task**
  - **Erfassung meteorologischer Meßwerte.**
- **Bus-Management-Task**
  - **Datenaustausch innerhalb des Systems: Information-Bus, Kommunikationsinterface**





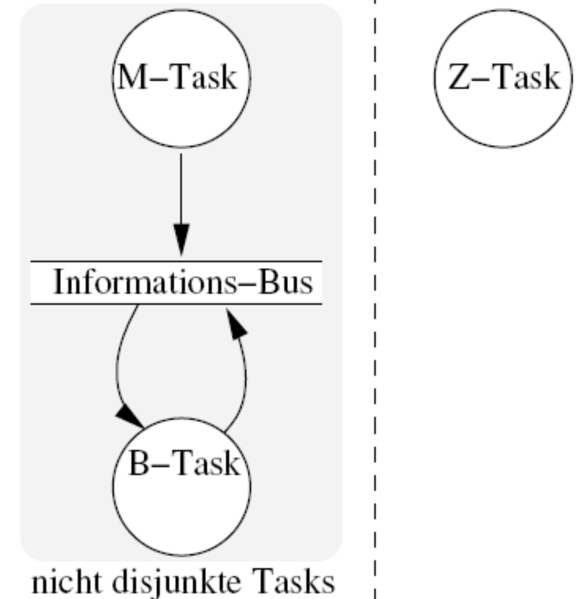
Programmierung

Kontrollfluß

Threads oder Prozesse, die (quasi-)parallel arbeiten

Disjunkte Prozesse

- **Ablauf eines Prozesses/Threads unabhängig von den anderen disjunkten Prozessen**



Nicht disjunkte Prozesse

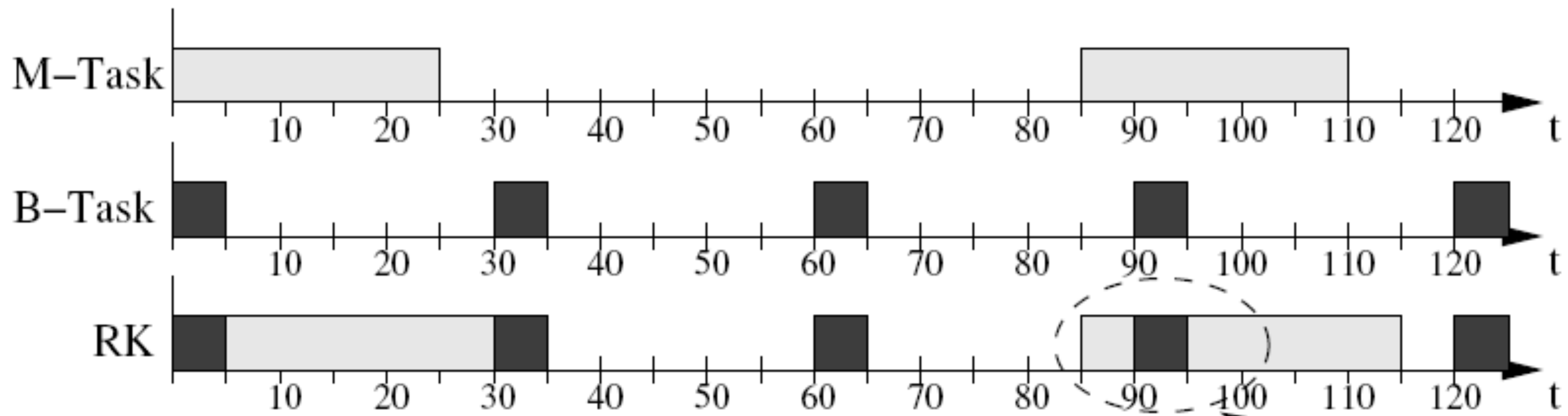
- **Konkurrierende Prozesse, die um den Zugriff auf Daten konkurrieren**
- **Kooperierende Prozesse (meist verkettet), ein Prozess liefert Daten für den anderen Prozess (Hersteller/Verbrauchermodell)**

Die Wirkung der gegenseitigen Beeinflussung nicht disjunkter paralleler Prozesse ist ohne Synchronisation nicht vorhersagbar und im Regelfall nicht reproduzierbar.

Programmierung

Kontrollfluß

Beispiel Pathfinder



Programmierung

Kritischer Bereich *(Critical Section)*

Programmteil, in dem auf gemeinsame Daten zugegriffen wird

Race condition: **Beim unsynchronisierten Zugriff mehrerer Threads bzw. Prozesse auf dieselben Daten:**

- **Ergebnis/Konsistenz hängt vom Prozessfortschritt ab**
- **Beispiel: Ringpuffer**

Race conditions unbedingt vermeiden

Höchstens ein Prozess tritt in einen kritischen Abschnitt ein

Mutex (mutual exclusion): gegenseitiger Ausschluß

Semaphore: Werkzeug zur Synchronisation

- **Signalisierung (Erzeuger-Verbraucher: Erzeuger gibt Sema, Verbraucher nimmt Sema)**
- **gegenseitiger Ausschluß (Sema nehmen bei Betreten, geben bei Verlassen eines kritischen Bereichs)**

Programmierung

Semaphore

Prinzipiell Integer Variable, die wie folgt verwendet wird:

- **Der Wert des Semaphors wird auf einen Maximalwert N initialisiert (soviele Prozesse dürfen kritischen Bereich betreten)**
- **P-Operation (Semaphore nehmen)**
  - **Bei einem Zugriff auf das Semaphor wird dessen Wert um 1 erniedrigt**
  - **der Prozess *schlafend* gelegt, wenn der neue Semaphor-Wert negativ ist**
- **V-Operation (Semaphore geben)**
  - **Bei der Freigabe einer Semaphore wird dessen Wert um 1 erhöht**
  - **falls der neue Semaphor-Wert kleiner gleich 0 ist, ein auf das Semaphor wartender (schlafender) Prozess aufgeweckt**

Originalbezeichnungen des „Erfinders“ Djikstra

P: (holländisch:) *passeeren (→ passieren)*

V: (holländisch:) *vrijgeven (→verlassen)*

Programmierung

Semaphore

- ▸ Theorie: Semaphore dient zur Steuerung des Zugriffs auf ein begrenzt verfügbares Betriebsmittel
- einfachster Fall: Betriebsmittel nur einmal vorhanden ▸nur eine Task (und ein Prozessor!) darf es gleichzeitig benutzen!! Struktur, mutex = mutual exclusion, P() und V() sind unteilbare Funktionen und müssen durch RBS realisiert werden (➔Multiprozessorsysteme sync mit anderen Prozessoren!!) ▸Vertagung der wartenden Tasks durch RBS ▸Warteschlange in der zeitlichen Ankunftsreihenfolge

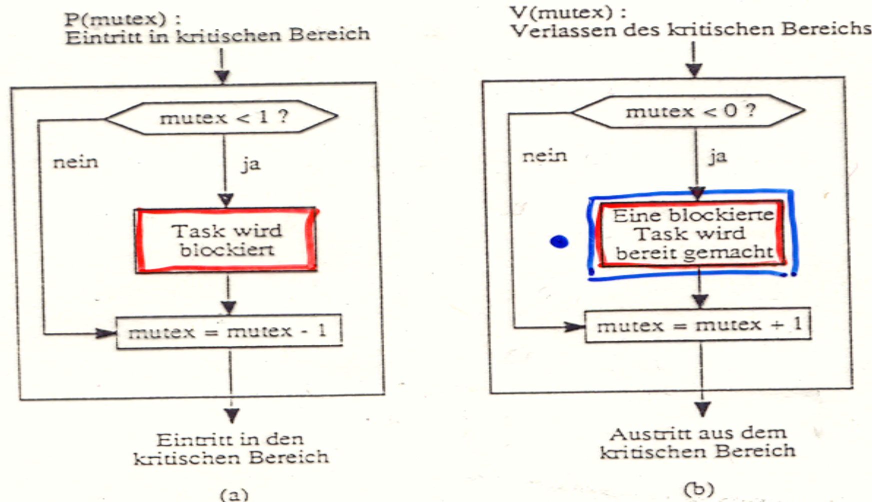


Abbildung 8.11: Die Anwendung der P-Operation (a) und der V-Operation (b) auf ein allgemeines Semaphor *mutex*. Diese Operationen sind nicht unterbrechbar was durch die Einrahmung angedeutet ist [Koch80]

P() + V()

! RBS - !

Funktion im Kernel

- Semaphor Queue
- Event absetzen Semaphore "frei" => prio-scheduling!

Programmierung

Semaphore pthread_mutex POSIX.1-2017

- **Ein Mutex ist eine Variable,**
  - **die einmal angelegt u. initialisiert wird (→ggf. im shared memory sichtbar über Taskgrenzen hinweg) und**
  - **von allen nutzenden Threads aus sichtbar sein muss,**
  - **die durch Funktionsaufrufe von Tasks/Threads belegt und wieder freigegeben wird.**
  - **Wird von einem anderen Thread versucht einen bereits belegten Mutex zu belegen, dann wird dieser automatisch vom System solange suspendiert bis der Mutex wieder freigegeben wird. Der nächste auf Belegung wartende Thread wird geweckt und gleichzeitig in einem atomaren Schritt wird der Mutex von diesem Thread belegt.**
- **Anlegen eines Mutex:**
  - **Definition einer Mutexvariable mit Defaultinitialisierung (frei):**
    pthread_mutex_t mutex* = *PTHREAD_MUTEX_INITIALIZER;
  - **Alternative Initialisierung durch Aufruf pthread_mutex_init() möglich.**
- **Belegen eines Mutex:**
  - **Mittels *int pthread_mutex_lock(pthread_mutex_t *mutex);* wird der Mutex mutex belegt, wenn er noch frei war ansonsten wird der Aufrufer solange suspendiert bis der Mutex wieder frei ist und alle vor ihm in der Warteschlange bedient wurden.**
- **Freigeben eines Mutex:**
  - **Mittels *int pthread_mutex_unlock(pthread_mutex_t *mutex);* kann der Mutex mutex von dem Thread, der vorher belegt hatte, wieder freigegeben werden.**
- **Testen und ggf. Belegen eines Mutex in einem Schritt:**
  - **Mittels int *pthread_mutex_trylock(pthread_mutex_t *mutex);* wird der Mutex mutex getestet ob er belegt ist, wenn er noch frei war wird er belegt, ansonsten wird zum Aufrufer mit entsprechender Fehlermeldung EBUSY zurückgekehrt; d.h. er wird nicht blockiert.**

Programmierung

Semaphore

- Bild : Synchronisation einer Erzeuger und einer Verbraucher Task!! ▸zwei Semaphore nötig: Leerung und Füllung, wiederum einfachster Fall: ein Platz zum Befüllen▸Erzeuger muss warten bis Verbraucher abgeholt hat
- ▸Praxisprobleme: durch Designfehler der SW ▸Verklemmungen!! beide warten aufeinander!!
- ▸Sonderfall: sog Monitore sind gemeinsam nutzbarer Code und Daten ▸sie werden den Tasks mittels Semaphore exclusiv zugeteilt. Will gleichzeitig ein anderer in den Monitor eintreten so muss er warten auf seine Freigabe

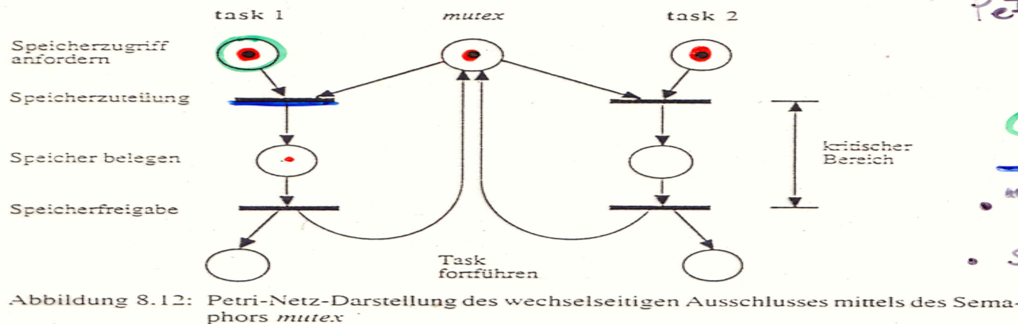


Abbildung 8.12: Petri-Netz-Darstellung des wechselseitigen Ausschlusses mittels des Semaphors *mutex*

Petri-Netz ≡ gerichteter Graph
„Marke“
Stelle
Transition
• „Marken kreisen“
• Schaltbdg. !

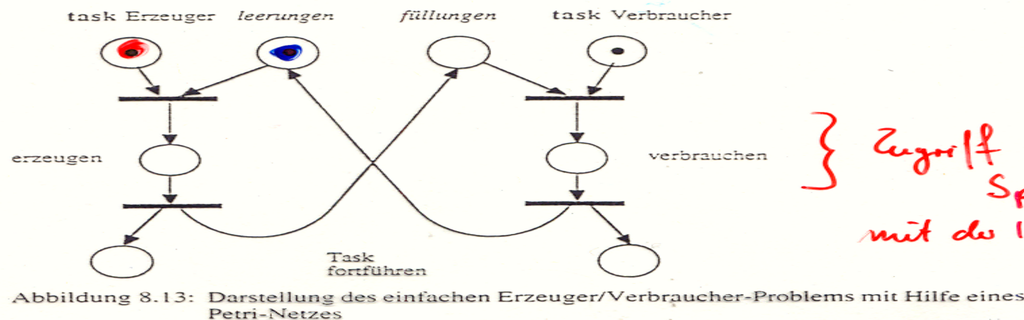


Abbildung 8.13: Darstellung des einfachen Erzeuger/Verbraucher-Problems mit Hilfe eines Petri-Netzes

} Zugriff auf gemeinsamen Speicher! mit der Kapazität 1 !

Programmierung

Semaphore OS9 mittels OS9-Events

- Semaphor-Operationen P() und V() müssen unteilbar in Einem ausgeführt werden. ➔ Signalisieren von und Warten auf Events ist eine unteilbare Operatiom. Die Anzahl der exclusiven Betriebsmittel, die es mittels eines Semaphors zu verwalten gilt, sei n.
- Lösungsidee:

  Anlegen eines Events z.B.: "Sema1"

  Vorbelegen des Event-Wertes mit n

  Winc auf -1

  Sinc auf 1
- ➔ Belegen P() :Der Benutzer will ein Betriebsmittel belegen und führt als P() einen *_os_ev_wait()*-Aufruf mit den Parametern Minval=1 und Maxval=n durch.

  ➔ Ist der Event-Wert im angegebenen Bereich -also eins der n Betriebsmittel frei- so kommt der Aufruf sofort zurück. Es wird aber Winc=-1 auf den Eventwert angewendet ➔ somit ist ein Betriebsmittel weniger frei.

  ➔ Ist der Eventwert nicht im Bereich, also 0, so wird der Aufrufer suspendiert.
- ➔ Freigeben V() : Der Benutzer gibt ein Betriebsmittel frei und führt als V() einen *_os_ev_signal()*-Aufruf

  durch. (*actv_flag = 0/1 ➔ egal weil winc nach dem erfolgten Wecken des ersten Wartenden angewendet wird und damit ggf. wieder 0 ist, und kein weiterer geweckt wird)*

  ➔ Sinc=1 wird einmal auf den Eventwert addiert

  ➔ War der Eventwert vorher 0 und es warten welche auf ein Betriebsmittel, so wird der

  Nächste geweckt
- !Achtung P() und V() müssen sich pärchenweise in jeder Applikation abwechseln!!
- Sollte der Eventwert kleiner 0 oder größer n werden, so liegt ein Fehler vor ➔ z.B. Überwachungstask bewacht den ungültigen Bereich und meldet Fehlbenutzung!!.

Programmierung

Kritischer Bereich

Beispiel Pathfinder

M–Task
B–Task
RK
S
1
0
–1
10 20 30 40 50 60 70 80 90 100 110 120 t

B–Task wird lauffähig, blockiert aber direkt an der P–Operation

M–Task = Task zur Erfassung meteorologischer Daten
B–Task = Bus–Management–Task
RK = Rechnerkernbelegung
S = Informationsbus–Semaphor

Programmierung

Kritischer Bereich

Beispiel

Pathfinder

Priorität

M–Task — niedrig

Z–Task — mittel

B–Task — hoch

RK

10 20 30 40 50 60 70 80 90 100 110 120 t

Deadline Verletzung

S 1 0 −1

M–Task blockiert

B–Task blockiert

M–Task = Task zur Erfassung meteorologischer Daten
Z–Task = Task zur Zustandserfassung
B–Task = Bus–Management–Task
RK = Rechnerkernbelegung
S = Informationsbus–Semaphor

Programmierung

Kritischer Bereich

Prioritätsinversion

Situation:

3 Tasks unterschiedlicher Priorität HOCH, MITTEL, NIEDRIG

- **HOCH und NIEDRIG haben gemeinsamen kritischen Bereich**
- **NIEDRIG ist im kritischen Bereich, wenn HOCH diesen ebenfalls betreten will**
- **HOCH wartet, bis NIEDRIG kritischen Bereich verlässt**
- **MITTEL verdrängt NIEDRIG, HOCH muss auf Beendigung von NIEDRIG warten**
- **MITTEL ist höherprior als HOCH !!!**

Programmierung

Kritischer Bereich

Prioritätsinversion, Lösung: Prioritätsvererbung

Prinzip, im Beispiel:

- **In dem Moment, in dem HOCH den kritischen Bereich betreten will, ist es wichtig, dass dieser wieder freigegeben wird. NIEDRIG ist aber dazu zu unwichtig.**
- **Also: In dieser Situation Priorität von HOCH auf NIEDRIG übertragen, bis NIEDRIG den kritischen Bereich verlässt**

Prioritätsvererbung, *Priority Inheritance Protocol (PIP)*

- **Falls ein Job HOCH eine Semaphore anfordert, die gegenwärtig von einem Job NIEDRIG mit niedrigerer Priorität gehalten wird, dann wird die Priorität von NIEDRIG auf die Priorität von HOCH angehoben.**
- **Sobald der Job NIEDRIG die Semaphore wieder freigibt, bekommt er seine initiale Priorität zurück.**

Programmierung

Kritischer Bereich

Prioritätsinversion, Lösung: Prioritätsvererbung

Alternativ: Priority Ceiling Protocol

- Anheben der Priorität auf die Priorität der höchstprioren Task, mit der er ein Betriebsmittel teilt (konkurriert)
- Kann Deadlocks (kommt gleich) ggf. verhindern
- Schwer implementierbar: Kritische Bereiche (insbesondere Tasks, die diese benutzen), müssen dem Scheduler bekannt sein
- Kaum verbreitet

Programmierung

Kritischer Bereich

Beispiel

Pathfinder

Priorität

M–Task — niedrig

Z–Task — mittel

B–Task — hoch

RK

10 20 30 40 50 60 70 80 90 100 110 120 t

Deadline Verletzung

S 1 0 −1 t

M–Task blockiert

B–Task blockiert

M–Task = Task zur Erfassung meteorologischer Daten
Z–Task = Task zur Zustandserfassung
B–Task = Bus–Management–Task
RK = Rechnerkernbelegung
S = Informationsbus–Semaphor

Programmierung

Kritischer Bereich

Beispiel Pathfinder

- **Priority Inheritance war abgeschaltet (wie von VxWorks aus Performancegründen empfohlen), wurde per Software-Upload aktiviert**

Priorität

M–Task niedrig

Z–Task mittel

B–Task hoch

RK

10 20 30 40 50 60 70 80 90 100 110 120 t

Prioritätsvererbung

S 1 0 −1 t

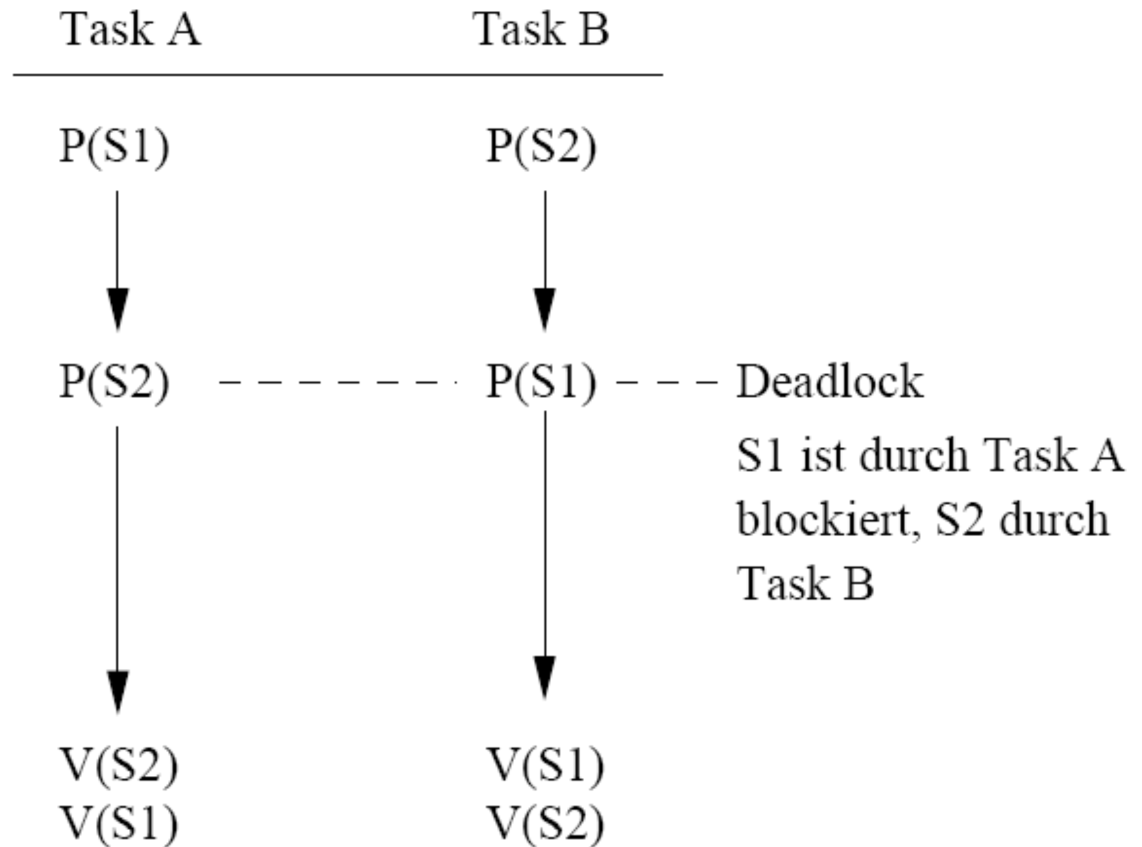
M–Task blockiert

B–Task blockiert

Programmierung

Deadlock

Critical sections **(gegenseitigem Ausschluß) können zu Verklemmungen (*Deadlocks*) führen**



Programmierung

Kritischer Bereich, Schreib-/Lese Locks

Bei Zugriff auf globale Daten

- **Race-Condition nur, falls zugreifende Rechenprozesse die Daten modifizieren**
- **Paralleles Lesen: unkritisch**

Methoden: Schreib-/Lese-Locks

- **Spezielle Semaphore**
  - **erlauben parallele Lesezugriffe**
  - **ermöglichen exklusiven Zugriff für schreibenden Job**
  - **Rechenprozess muss bei der Anforderung des Mutex mitteilen, ob er den kritischen Abschnitt nur zum Lesen oder auch zum Schreiben betreten möchte**

Programmierung

Kritischer Bereich, Schreib-/Lese Locks

Fälle:

1. **Der kritische Abschnitt ist frei, ein Rechenprozess möchte zugreifen**
   - **Der Zugriff wird gewährt**
2. **Ein lesender Rechenprozess belegt den kritische Abschnitt**
   - **Ein Prozess, der schreiben möchte, wird blockiert.**
   - **Ein Prozess, der lesen möchte, bekommt Zugriff.**
3. **Ein schreibender Rechenprozess belegt den kritische Abschnitt**
   - **Der anfragende Rechenprozess wird blockiert**

Programmierung

Kritischer Bereich, Weitere Schutzmaßnahmen

User-Prozesse: Semaphor zum Schutz kritischer Abschnitte

Betriebssystemkern, Treiber, Interrupt-Service-Routinen???

- **Unterbrechungssperre**
  - **Interrupts für die Zeit des Zugriffs auf den kritischen Abschnitt sperren**
  - **Latenzzeiten kurz halten!!!**
- **Spinlocks.**
  - **Bei Multiprozessorsystemen können zwei oder mehr Interruptserviceroutinen real parallel bearbeitet werden, lokale Unterbrechungssperre hilft nicht weiter**
  - **Gemeinsame Variable entscheidet, ob ein kritischer Abschnitt betreten werden darf**
  - **Busy waiting, falls kritischer Abschnitt bereits besetzt**

Methode	User Level	UP: Kernel Level	SMP: Kernel Level
Unterbrechungssperre	Nein	Ja	Nein
Spinlocks	Nein	Nein	Ja
Semaphore	Ja	Ja	Ja

Events

Events: Synchronisation bezüglich des Programmablaufs zweier Tasks
Eine Task wartet auf ein Event (Ereignis)
Andere Task setzt/signalisiert Event
häufig mit Semaphore oder MessageQueue realisiert

Schönes Konzept:

Condition variables: Event, das an die Änderung von Bedingungen (z.B. Überschreiten eines Puffer-Füllstands) geknüpft ist.

Dabei gehören (immer) 3 Dinge zusammen:

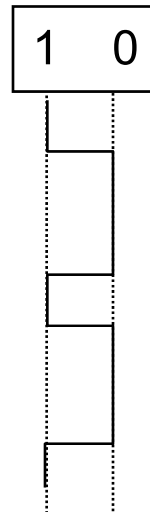
- **Condition variable (entspricht dem Event, wenn sich die Bedingung ändert)**
- **Globale (Shared) Variablen, über die eine Bedingung (Condition) formuliert ist**
- **Mutex, die den Bereich „Überprüfen der Bedingung – Warten auf Veränderungssignalisierung“ atomar (kritischer Bereich!!!) macht**

Programmierung

Condition Variables, Beispiel

```
pthread_mutex_t spar_lock;
pthread_cond_t spar_aenderung;
unsigned int erspartes;

void entnimm_erspartes(int wieviel)
    {
 pthread_mutex_lock(&spar_lock);
 while (erspartes < wieviel) {
  pthread_cond_wait(&spar_aenderung,
                    &spar_lock);
 }
 erspartes -= wieviel;
 pthread_mutex_unlock(&spar_lock);
}
```



Mutex freigeben, solange Thread wartet

```
void spar_was(int wieviel) {
 pthread_mutex_lock(&spar_lock); erspartes += wieviel;
 pthread_cond_broadcast(&spar_aenderung);
 pthread_mutex_unlock(&spar_lock);
}
```

Programmierung

Condition Variables, POSIX.1-2017 API

int pthread_cond_init(pthread_cond_t * *cond* , pthread_condaddr_t * *cond_attr*);

int pthread_cond_signal(pthread_cond_t * *cond*);

Ereignis verschicken. Genau eine auf das Ereignis wartende Task aufwecken

int pthread_cond_broadcast(pthread_cond_t * *cond*);

Ereignis an alle Tasks schicken, die auf das Ereignis warten

int pthread_cond_wait(pthread_cond_t * *cond* , pthread_mutex_t * *mutex*);

Gibt in einer atomaren Operation die mit dem Event assoziierte Mutex frei. Die aufrufende Task blockiert. Mutex wird beim Aufwachen wieder belegt.

int pthread_cond_timedwait(pthread_cond_t * *cond* , pthread_mutex_t * *mutex* , const struct timespec * *abstime*);

int pthread_cond_destroy(pthread_cond_t * *cond*);

Programmierung

OS9 Events (1)

- Jedes Event muss vor Verwendung vom Benutzer explizit unter Vergabe eines systemweit eindeutigen Eventnamens angelegt werden (alle Events sind global bekannt).
- Ein Event ist ein vorzeichenbehafteter 32-bit Zahlenwert.
- Jeder ist berechtigt jedes existierende Event zu benutzen oder eins zu erzeugen.
- Eventbezogen kann auf das Auftreten eines bestimmten Zahlenwertes oder auch auf das Erreichen eines vorgebbaren Wertebereichs des Zahlenwertes gewartet werden (➔event-queue).
- Die programmgesteuerte Veränderung des Datenwertes eines Events entspricht logisch dem Setzen des Events ▸Der veränderte Datenwert wird vom OS9 mit den Wartebedingungen von Tasks auf dieses Event verglichen und diese ggf. rescheduled.

OS9 Events (2)

- Alle angelegten Events werden systemweit in einer doppelt verketteten Liste vom Betriebssystem verwaltet. Ein Listenelement ist im wesentlichen wie folgt aufgebaut:

Event ID (32-bit)
Länge des Eventnamens(16 bit) **Zeiger auf Eventname**
Linkcount (links)
Zugriffsrechte (MP_OWNER_READ | MP_OWNER_WRITE |MP_GROUP_READ | MP_GROUP_WRITE |MP_WORLD_READ | MP_WORLD_WRITE)
OwnerID
Waitinkrement (Winc 16-bit)
Signalisierungsinkrement (Sinc 16bit)
Zahlenwert (32-bit)
Pointer to next event in event-list (next)
Pointer to previous event in event-list (prev)

Programmierung

OS9 Events (3)

- Ein Event wird mittels *_os_ev_creat()* erzeugt. Damit auch andere Tasks dieses Event verwenden können vergibt der Erzeuger einen systemweit eindeutigen Eventnamen unter dem es u.a. auch auffindbar ist.
- ▸Aus Geschwindigkeitsgründen wird neben dem Namen noch eine systemweit eindeutige EventID geführt, die es dem Betriebssystem gestattet ohne String-Vergleich auf das Event zuzugreifen.
- ▸Alle weiteren Dienste zur Eventnutzung benutzen daher als Übergabeparameter die EventID anstelle des Eventnamens.
- ▸Beim Anlegen des Events initialisiert der Benutzer den Zahlenwert des Events
- ▸Winc ist das sog. Waitinkrement. Wird eine Task, die auf einen Eventzustand wartete, geweckt so wird Winc auf den Zahlenwert des Events addiert.
- ▸Sinc ist das sog. Signalisierungsinkrement. ´Setzt´ eine Task mittels SYSA das Event so wird der Wert Sinc auf den Zahlenwert des Events addiert. Anschließend wird, falls gewünscht, mit dem neuen Wert die Event-Warteliste durchgesehen ob eine Task zu wecken ist.
- ▸Der Linkcount wird vom OS9 verwaltet und angelegt. Er gibt an wieviele Tasks dieses Event aktuell benutzen. Nur Events mit Linkcount = 0 können wieder aus dem System entfernt werden. Beim Erzeugen des Events wird der Linkcount auf eins gesetzt.
- ▸Die beiden Pointer next und prev dienen zu Verkettung der Events zu einer Liste und werden von OS9 angelegt und verwaltet.
- ▸Das Event wird unter der Benutzerkennung des erzeugenden Programms angelegt. Damit jeder mit dem Event arbeiten kann, müssen die Zugriffsrechte beim Erzeugen des Events entsprechend gesetzt werden.

Programmierung

OS9 Events (4)

- *Beispiel Eventerzeugung:*

```
#include <events.h>
#include <types.h>                    ‣Includes für alle Event-Calls
#include <const.h>
#include <memory.h>
int32      winc,sinc,value;
error_code   myerr;
event_id    id_pdv_event;
static char event_name [] = "PDV_event";
winc = 0; /* wait-inkrement soll nicht sein */
sinc = 1; /* jedes ´setzen´ erhöht den Wert um eins*/
value = 0; /* Startwert des Zahlenwertes */
u_in32 perm = MP_OWNER_READ  | MP_OWNER_WRITE |MP_GROUP_READ |
                                MP_GROUP_WRITE |MP_WORLD_READ | MP_WORLD_WRITE ;
myerr  = _os_ev_creat ( winc,sinc,perm,&id_pdv_event,event_name,value,MEM_ANY);
if ( myerr != SUCCESS ) { /*Fehlerbehandlung */ }
```

‣*bei Rückkehr wird in id_pdv_event die EventID abgespeichert*
‣*Weitere SYSAs nutzen diese ID*
‣*MEM_ANY gibt OS9 die Möglichkeit dieses Event in einem der möglichen Speicherbereiche anzulegen, unter x86 nicht vorhanden.*

Programmierung

OS9 Events (5)

- **Anlinken an ein existierendes Event mittels Namen durch *_os_ev_link ()* ▸*Linkcount um eins erhöht*:**
  event_id id_pdv_event; /*▸enthält nach Aufruf die EventID */
  static char event_name [ ] = "PDV_event";
  error_code _os_ev_link (event_name, & id_pdv_event);

- **Event setzten: *_os_ev_set()* oder *_os_ev_signal()* ➔beide verwenden EventID ➔ggf. anlinken nötig**

- **▸Signalisiere Event *_os_ev_signal* (*event_id ev_id, int32 *value, u_int32 actv_flag*)**
  Ablaufschritte:
  1. **Das Event *ev_id* wird gesucht.**
  2. **neuer Zahlenwert N des Events = alter Zahlenwert A + Sinc.**
  3. **Die nächste in der Reihe wartende Task W, die auf den Zahlenwert N wartet wird aktiviert.**
  4. **Winc wird zum Zahlenwert N des Events dazuaddiert wenn diese Task W gefunden und geweckt wurde:**
  5. **neuer Zahlenwert N = Zahlenwert N + Winc**

  5.a. Wenn *actv_flag* == 1 und eine Task aktiviert wurde gehe zu Schritt 3 sonst 6.

  6. **Der Zahlenwert N des Events wird in *value* gespeichert.**

  ▸Wahlweise werden ggf. nur die erste Task oder alle aktiviert!!
  ▸Die Ausführung der Winc - Addition löst weitere Aktion aus wenn actv_flag = 1 !!

Programmierung

OS9 Events (6)

- ▸Signalisiere und setze den Zahlenwert eines Event auf einen absoluten Wert

  *_os_ev_set (event_id ev_id, int32 *value, u_int32 actv_flag)*

  Ablauf: wie *_os_ev_signal()* bis auf Schritt 2 ▸Anstelle Sinc wird der Zahlenwert N des Events auf den übergebenen *value* Wert gesetzt.
- Event freigeben : *_os_ev_unlink (event_id ev_id); reduziert den Linkcount um eins*
- Event löschen: *_os_ev_delete (char*event_name);* wenn Linkcount 0 ist ➔löschen des Events

- Warten auf das Auftreten eines Eventereignisses *_os_ev_wait()/_os_ev_waitr()* ▸vorheriges Anlinken des Events nötig!

Ablauf: *_os_ev_wait (event_id ev_id, int32* value, signal_code* si, int32 min_val, int32 max_val);*

1. Zahlenwert (Z) des Events *ev_id* wird verglichen:
   a. Falls *min_val* <= Z <= *max_val erfüllt ist, wird Winc addiert* und *Rückkehr aus _os_ev_wait().*
   b. Sonst: Task wird in die Warteschlange (event queue) eingereiht
2. Späterer Zeitpunkt
   ▸Event wird von anderer Task gesetzt und nun sei der gesetzte Zahlenwert (Z) im Wartebereich
   ▸Die erste Task die auf den Zahlenwert in der Queue wartete wird geweckt, in die aktive Queue eingereiht und Winc addiert

!!! Ausnahme: ▸Die Task wird auch durch Empfang eines ´Signals´ geweckt
▸Test der Ursache des Weckens nötig!!!

Programmierung

OS9 Events (7)

Bei Rückkehr wird in *value* der Eventwert gespeichert, der zum Weckzeitpunkt bestand:

- **wenn nun ein Signal die Weckursache war, muss eine Signalinterceptroutine existieren.**

  Dann liegt der Eventwert bei Rückkehr aus *_os_ev_wait()* nicht im erwarteten Zahlenbereich (*min_val,max_val*).

  - **In *signal_code si* wird im Falle Weckens durch einen Signalempfang die letzte empfangene Signalnummer hinterlegt.**

- ***_os_ev_waitr()* ‣wartet auf das Erreichen eines Wertes relativ (Offset) zum gerade gültigen Wert des Events**

- ***OS9-Shellkommando events (Display Active System Events)* → events dient zur Anzeige der im System befindlichen events.**

  $ events

Event ID	Owner	Perm	Value	W-inc	S-inc	Links	Name
---------	-------	----	----------	------	------	------	------------
00010001	0.0	0003	0	0	0	1	sysmbuf
00030003	0.0	0003	0	-1	1	1	spf_rx

- **$ events –h ==> Anzeige Value in hex**
- **$ events –k=name ==> Killen des Events name**

Programmierung

Signale

Software-Interrupts auf Applikationsebene

Signal führt zu einer Unterbrechung des Programmablaufs innerhalb der Applikation

Programm

- **wird abgebrochen (`exit()` im Default-Signalhandler) oder**
- **reagiert mit einem vom Programm zur Verfügung gestellten Signal-Handler (ähnlich einer Interrupt-Service-Routine).**

Signale können ausgelöst werden

- **durch eine Applikation**
- **durch Ereignisse innerhalb des Betriebssystem selbst.**
  - **Zugriff auf einen nicht vorhanden Speicherbereich: dem Prozess wird ein Segmentation-Fault-Signal zugeschickt**
  - **Könnte abgefangen werden, z.B. um wichtige Daten noch zu sichern**

Signale

Unterschiede Signale/Events

Signals	Events
Signals kommen asynchron zum Programmablauf und werden asynchron verarbeitet.	Events kommen synchron zum Programmablauf und werden synchron verarbeitet.
Charakter einer Interrupt-Service-Routine (Software-Interrupt).	Rendezvous-Charakter

Programmierung

Signale

Warnung!!!

Ein Signal führt i.d.R. zum sofortigen Abbruch eines gerade aktiven, blockierenden Systemcalls (<>OS9!!).

Werden im Rahmen einer Applikation Signale verwendet bzw. abgefangen, sollte jeder Systemcall daraufhin überprüft werden, ob selbiger durch ein Signal unterbrochen wurde und, falls dieses zutrifft, muß ggf. der Systemcall *neu* aufgesetzt werden!

Falsch (z.B. Linux)

```
read ( fd, &buf, buflen)
```

Kehrt bei Signal zurück, ohne Fehleranalyse: In buf steht noch der alte Wert!!!

Richtig (z.B. Linux)

```
while ((ret = read ( fd, &buf, buflen)) < 0 && errno == EINTR);
```

Programmierung

Signale, POSIX.1-2017 API

```
int sigaction( int signum,
                    const struct sigaction * act ,
                  struct sigaction * oldact );
```

Mit diesem Systemcall kann das Verhalten (Aktion) parametriert werden, das eine Task bei Erhalt eines spezifischen Signals durchführt.

`act` **enthält Adresse des Signal-Handlers, der bei Erhalt des Signals vom Betriebssystem aufgerufen wird**

```
int kill( pid_t pid , int sig );
```

Einer Task oder einer Task-Gruppe ein Signal schicken

Programmierung

OS9 Signale (1)

- **Signale sind sog. Software-Interrupts die programmgesteuert von einer auslösenden Task an eine oder alle Anwendungstasks gesendet werden können.**
- **Ein Signal besteht aus Signalnummer (signal-code) und EmpfängertaskID.**
- **OS9000-Signalnummern sind systemweit gültig und sind wie folgt belegt:**

1	**Wake-up signal**. Sleeping/waiting processes receiving this signal are awakened, but the signal is not intercepted by the intercept handler. Active processes ignore this signal. A program can receive a wake-up signal safely without an intercept handler. The wake-up signal is not queued.
2	**Keyboard abort signal**. When <control>E is typed, this signal is sent to the last process to perform I/O on the terminal. Usually, the intercept routine performs exit(2) when it receives a keyboard abort signal
3	**Keyboard interrupt signal**. When <control>C is typed, this signal is sent to the last process to perform I/O on the terminal. Usually, the intercept routine performs exit(3) when it receives a keyboard interrupt signal.
4	**Unconditional system abort signal**. The super user can send the signal to any process, but non-super users can send this signal only to processes with their group and user IDs. This signal terminates the receiving process, regardless of the state of its signal mask, and is not intercepted by the intercept handler.
5	**Hang-up signal**. SCF sends this signal when the modem connection is lost.
6-19	Reserved
20-25	Reserved
26-31	User-definable signals that are deadly to I/O operations.
32-127	Reserved
128-191	Reserved
192-255	Reserved
256- 4294967295	User-definable non-deadly to I/O signals.

Programmierung

OS9 Signale (2)

- Signal 0 kann mit dem OS9000-Kommando kill <EmpfID> an die eigenen Tasks (group,owner) verschickt werden.
- ▸OS9000: signalcode 0 ▸nur an Tasks mit gleicher group/user Id ➔Ausnahme SuperUser 0,0 darf an alle
- Signal 2/3 wird durch Eingabe ^e und ^c durch den SCF-Devicetreiber erzeugt und an die Task geschickt die zuletzt eine Ausgabe am Terminal getätigt hat!!
- Signal 1 hat eine Ausnahmefunktion! Signal 1 ist das WAKE-UP Signal und führt dazu dass die betreffende Empfängertask geweckt wird. !!Achtung in diesem Fall wird die Signal-Empfangsroutine nicht durchlaufen!!
- Ab Signal 256 kann jeder Benutzer eigene Signale belegen.
- Signal 4 wird nicht zugestellt und kann von einer User-State Task nur an Tasks der eigenen Group/Owner ID geschickt werden. Dann terminiert die Empfängertask. Super-User Tasks können an alle schicken.
- OS9: Existiert keine Signal-Empfangsroutine und wird ein Signal geschickt, so terminieren user-mode Tasks/system-mode Tasks nicht. (Ausnahme: Gilt nicht für signal 1!)
- ➔OS9000: Broadcast ein Signal an mehrere ▸Destinationtask pid =0 ▸Signal an alle mit gleicher user/groupID
- Damit die Empfängertask ein Signal empfangen kann, stellt sie eine Signal-Empfangsroutine bereit und läßt diese vom Betriebssystem als sog signal-intercept-Routine registrieren.

Programmierung

OS9 Signale (3)

- Signal senden‣aktive Task verschickt ein Signal
- ‣das RBS führt ohne Taskwechsel die registrierte Signalempfangsroutine des Empfängers sofort aus.
  ‣Die signal-intercept-Routine des Empfängers sollte demnach so kurz wie möglich sein.
- ‣die angesprungene Task wertet in der Interceptroutine den ihr übergebenen Signalcode aus und setzt im einfachsten Fall eine eigene globale-statische Variable (➔volatile deklariert, damit keine Compiler-Optimierungen wg. Registerverwendung!!) damit nach Verlassen der Interceptroutine der Wert noch vorhanden ist. ‣Rückkehr aus Interceptroutine besonders: _os_rte()!! ‣Rückkehr zur Sendertask
- Fall 1: Die bereite Empfangstask befindet sich in diesem Augenblick irgendwo im Taskcode und wartet nicht explizit auf das Auftreten eines Signals. (programmgesteuerte Synchronisation)

  ‣Wenn die Task an die Reihe kommt (scheduled wird) kann programmgesteuert vom Task-Programmcode (außerhalb der signal-Interceptroutine) z.B. die globale Variable abgefragt und verarbeitet werden.

  ‣Programmgesteuerte Abfragemethode ‣Ort und Zeit der Abfrage in der Hand des Programmierers

  ‣ Nachteil im einfachsten Fall einer globale Variable: typischerweise sind mehrfache Abfragen nacheinander im Programmcode eingestreut ‣ggf. Verlust eines gleichen Signals welches kurz hintereinander wiederholt auftritt durch zu großen Zeitabstand zwischen zwei Abfragen ein und derselben Variablen.

Programmierung

OS9 Signale (4)

- Fall 2: Die Empfangstask hatte sich mittels _os_sleep()-Aufruf suspendiert und die Schlafdauer (timeout) ist noch nicht abgelaufen. *(Beachte: Der Aufruf von _os_sleep() setzt die Signalempfangsmaske auf NULL und erlaubt somit sofort den Empfang von Signalen)*
  - ▸ OS9: Sie ruht in der sog sleep-Queue, die nach der Restschlafdauer -kürzeste zuerst- sortiert ist, und wartet auf das Ablaufen ihres einstellbaren sleep-Timers. (Timerwert=0 ▸warten unendlich)
  - ▸OS9: Empfängt eine Task in der sleep-Queue ein Signal oder läuft ihr Timer ab so wird sie automatisch sofort in den Zustand bereit versetzt. Ihre Priorität bestimmt ihre Rangstufe in der Bereit-Warteschlange.
  - ▸OS9: Die Task wird nach _os_sleep()-timer Aufruf fortgesetzt ▸wenn der zurückgegebene Zeitwert ungleich 0 ist, so wurde sie durch ein Signalempfang geweckt. ▸Empfangener Signalcode wird mit zurückgegeben.

- ▸OS9: Um kritische Abschnitte im Taskcode gegen das Stören durch Signalunterbrechungen zu schützen gibt es vergleichbar der Interruptmaske eine Signalempfangsmaske. Ist sie Null so ist der Signalempfang möglich.
  - ➔_os_sigmask(1) erhöht die Signalempfangsmaske um Eins. Ist danach die Signalmaske grösser gleich eins, so ist der Signalempfang gesperrt;
  - ➔_os_sigmask(-1) erniedrigt die Sigmalempfangsmaske um eins, wenn diese nun Null ist, ist der Signalempfang wieder möglich
  - ➔ die Aufrufe _os_sigmask(0) oder ein Aufruf von _os_sleep() setzen die Signalempfangsmaske auf den Wert Null und entsperren für den uneingeschränkten Signalempfang .

Programmierung

OS9 Signale (5)

- ➔ Ist der Signalempfang in einer Task durch den Wert Null in der Signalempfangsmaske und das Vorhandensein einer registrierten Signalintercept möglich, so führt OS9 folgende Schritte aus:

OS9:	Vorbereiten des Stacks und der Aufrufumgebung der zu rufenden Taskumgebung
OS9:	*_os_sigmask(1)*
OS9:	Aufruf der Taskinterceptroutine der Task, erster und einziger Parameter die Signalnummer
Taskintercept:	Die Taskinteceptroutine beendet sich mit *_os_rte()*
OS9:	In der Endebehandlung durch OS9 Aufruf *_os_sigmask(-1)*
OS9:	Aufräumen des Stacks und Rückkehr zu vorherigem Zustand

Beachte: Wird in der Signalinterceptroutine *_os_sleep() oder sigmask(0)* aufgerufen, so wird die Signalempfangsmask auf Null gesetzt und im Signalempfangsvorgang ist der eigene rekursive Aufruf innerhalb der Signalinterceptroutine erlaubt. Die Rekursivität im eigenen Aufruf ist in diesem Fall durch die eigene Implementation in der Signalinterceptroutine zu berücksichtigen

Hinweis: Im oben genannten Fall 2 ist man nach Rückkehr aus dem durch den Signalempfang geweckten *_os_sleep()* bereit für den nächsten Signalempfang, da *_os_sleep()* die Signalmaske auf NULL setzt. Soll bei der Rückkehr aus *_os_sleep()* aber der Signalempfang gesperrt sein, so kann man in der eigenen Signalintercept-Routine mittels des Aufrufs *_os_sigmask(1)* die Signalempfangsmask um eine weitere Eins erhöhen, so dass durch das *_os_rte() und der Ausführung von sigmask(-1) durch OS9* die Signalmaske grösser Null ist und damit weiterhin gegen weiteren Signalempfang gesperrt bleibt.

- ▸Signale werden sequentiell an die Empfänger weitergereicht. ➔für jedes Signal wird je einmal die Signal-Interceptroutine aufgerufen.
- ▸Ob gesperrt oder nicht, unbearbeitete Signale werden in der Reihenfolge ihres Auftretens gequeued.

Programmierung

OS9 Signale (6)

- *Beispiel Empfangstask:*

```
volatile signal_code Empfangs_signal;
void  signal_empfang (signal_code signal)
{
Empfangs_signal = signal;
_os_rte();
}
main (int argc, char* argv[ ])
{
u_int32 zero; error_code myerr; signal_code si ;
if ( (myerr =_os_intercept ( signal_empfang, _glob_data) ) !=SUCCESS) exit (myerr);
 /* Registrierung der Intercept_Routine*/
zero = 0 /* unendlich lang schlafen */ ;
if ( (myerr =_os_sleep(&zero,&si) ) !=SUCCESS) exit (myerr);
                    /* hier: warten zero=0 unendlich auf ein Signal !!Standardrückkehrwert:
                      ‣zero ist verändert und enthält Restzeit bis Ablauf Timer */
printf ("Signal empfangen: %d oder %d \n",Empfangs_signal,si); /* welches war´s denn? */
}
```

- *Beispiel Sendetask: (includes wie oben)*

```
if ( (myerr =_os_send ( (process_id) EmpfangsPID, (signal_code) Signalnummer)) !=SUCCESS) exit (myerr);
```

Programmierung

OS9 Signale (7)

- **Timer-Alarme sind eine Sonderform der Nutzung des Signalmechanismus in Verbindung mit einer Uhr. Anstelle einer Signalsendetask schickt das RBS nach einer einstellbaren Zeit ein vorher vereinbartes Signal an eine Task (auch periodisch).**
- **OS9: Ein Alarm ist an eine Uhr gebunden. Nach Ablauf einer einstellbaren Uhrzeit erhält der Alarmauftraggeber ein frei vorgebbaren Signalcode gesandt. Alarmuhrzeiten können Absolutzeiten, Deltazeiten und auch relative Zeiten sein. Es sind auch zyklische Alarme möglich▸Watchdog ähnlich.**

  Alle Alarme benötigen: #include <alarm.h>

- **Zyklischer Alarm:** ***error_code _os_alarm_cycle (alarm_id *alrm_id, signal_code signal,u_int32 time);***
  signal wird nach time (i.d.R. in TICs) periodisch an die eigene Task geschickt
  alarm_id dient zur Identifikation dieses Alarms.
- **Einmal Alarm:** ***error_code _os_alarm_set (alarm_id *alrm_id, signal_code signal,u_int32 time);***
  signal wird nach time (i.d.R. in TICs) einmal an die eigene Task geschickt
  alarm_id dient zur Identifikation dieses Alarms.
- **Alarm Löschen:** ***error_code _os_alarm_delete (alarm_id alrm_id);***
  Der Alarm ***alarm_id wird gelöscht, damit wird kein Alarmidentifiziert durch alarm_id mehr geschickt.***

Programmierung

Datenfluss/Inter-Prozess-Kommunikation

Methoden, um Daten zwischen Tasks zu transportieren (Inter-Prozess-Kommunikation, IPC):

- **Mailbox/Messages**
- **Shared-Memory**
- **Sockets**

Programmierung

Datenfluß/Inter-Prozess-Kommunikation

Message Queues

- Datenpakete mit unterschiedlicher, aber nach oben begrenzter Größe von Task A nach Task B senden
- Message Queues per Namen über das Filesystem referenziert, task-intern per Message Queue Id identifiziert
- Blockierendes Senden (Queue voll) und Empfangen (nichts in der Queue)

Programmierung

Inter-Prozess-Kommunikation/Message Queue POSIX API

```
typedef struct {
  long mq_maxmsg;   /* maximum number of messages */
  long mq_msgsize;  /* maximum message size */
  ...
} mq_attr;
```

```
mqd_t mq_open(const char *name, int oflag [, mode_t mode, mq_attr *attr]);
```

Öffnen einer systemweit sichtbaren benamten Message Queue. `Mode` und `attr`: wenn MQ erzeugt wird z.B. name = "/mymessagequeue"

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int
    msg_prio);
```

Message *msg_ptr* mit gegebener Länge *msg_len* versenden, Messagequeuesortierung absteigend nach Priorität und bei gleicher Priorität alt vor neu. Default blockierend wenn Queue voll.

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int
    *msg_prio);
```

Message im Puffer *msg_ptr* empfangen, nicht mehr als *msg_len* Zeichen. Default blockierend wenn Queue leer.

Programmierung

PIPES I (hier: Implementation OS9)

- Pipes dienen primär zur unidirektionalen Daten-Kommunikation zwischen zwei oder mehr Prozessen.
- Ein oder mehrere Prozesse sind Sender und ein Prozess der Empfänger. Das RBS OS9 übernimmt in einer FIFO-Queue, genannt Pipe, die Zwischenspeicherung der Daten.
- Es gibt unnamed Pipes und named Pipes.
- Unnamed Pipes werden immer dann von OS9 angelegt wenn mehrere Kommandos mittels ! verkettet werden. ▸die Ausgabe des ersten Kommandos gelangt in ein unnamed Pipe von der das nach dem ! folgende Kommando die Daten als Eingabe einliest.
- Named Pipes tragen wie eine Datei einen Namen und sind bei OS9 auf dem Device /pipe abgelegt. Dieses Device läßt sich mittels der normalen Dateikommandos (dir -e /pipe ,list...) ansprechen. Auf dem Device /pipe existiert kein hierarchisches Directorysystem. Alle named-Pipes liegen direkt im sog. Root-Directory. Anstelle der Größe der Pipes wird beim *dir -e /pipes*-Kommando die Anzahl der im FIFO befindlichen Bytes angegeben.
- Named//Unnamed Pipes in OS9 sind voreingestellt maximal 128 Byte groß ▸programmgesteuert kann aber beim Anlegen von named-Pipes die maximale Größe frei gewählt werden.

Programmierung

PIPES II

- *Beispiel: Anlegen,Beschreiben und Anzeigen einer named-Pipe mittels mshell-Kommandos*

```
$dir -e /pipe
        Directory of /pipe  09:49:29
Owner   Last modified      Attributes    Sector    Bytecount    Name
--------  ------------------      -----------     --------     ------------     -------

$echo   "Testdaten fuer die Named-Pipe hugo!!"  >/pipe/hugo
$dir -e  /pipe
        Directory of /pipe  09:50:10

Owner   Last modified      Attributes           Sector    Bytecount    Name
--------  ------------------      ----------------        --------     ------------     -------
0.0     03/10/01 09:50     --------------wr      372780           37      hugo

$list  /pipe/hugo
Testdaten fuer die Named-Pipe hugo!!
$dir -e /pipe
                                Directory of /pipe  09:51:00
Owner   Last modified      Attributes    Sector    Bytecount    Name
--------  ------------------      -----------     --------     ------------     -------
$
```

Programmierung

PIPES III

- Pipes sind Dateien die eine variable Länge haben▸Der Sender und der Empfänger behandeln Pipes mit denselben Betriebssystemaufrufen wie normale Dateien. ▸Eine Pipe muss, wie jede Datei, erzeugt/geöffnet/geschlossen werden ▸Filename: "/pipe/<mein_pipe_name>" ▸Es werden die üblichen Lese/Schreibaufrufe für Dateien verwendet ▸Es gibt keinen Fileposition-Zeiger ▸Der Sender schreibt beliebig in die Queue; die Daten werden an die Queue angehängt. ▸Der Empfänger liest aus der Queue aus, wobei das älteste Datum zuerst gelesen wird.
- Pipes, die leer sind und die von keinem Prozess geöffnet (benutzt) sind, werden von OS9 gelöscht!!!
- Named-Pipes unterliegen der Zugriffsrechtsvergabe wie alle Dateien.
- Besonderheiten beim Schreiben bzw. Lesen von named-Pipes:
  - Schreiben auf eine volle Pipe ▸Sender wird suspendiert bis wieder Platz ist (➔Signalmechanismus von OS9 wird genutzt !! Vorsicht es darf kein sigmask(1) gesetzt sein!!)
  - Lesen von einer leeren Pipe ▸Empfänger erhält EOF-Fehler
  - Lesen wenn zu wenig Daten da, aber kein! EOF im Puffer ▸Empfänger blockiert und wartet auf nächste Zeichen
  - Lesen wenn zu wenig Daten da, aber EOF in der Pipe ▸Empfänger erhält alle verfügbaren Daten und Anzahl der tatsächlich gelesenen Daten < Anforderung zurück aber kein EOF-Fehler.
- Öffnen einer nicht existierenden Pipe ▸Fehlermeldung
- Alle Benutzer schließen den Zugriff zu einer named-Pipe ▸wenn noch Daten in der Pipe, dann bleibt sie bestehen, ansonsten wird sie gelöscht.

Programmierung

PIPES IV

- Programmgesteuertes Anlegen/Öffnen und Schließen einer named-Pipe: *_os_create(),_os_open(),_os_close()*
- *Create:*

```
path_id pd;
u_int32 size = 1000;  /* Maximale Pipegröße statt 128 Byte nun 1000 ‣S_ISIZE zeigt im create an,
dass size als Parameter ausgewertet werden soll!! */
err = _os_create ("/pipe/meine_pipe", S_ISIZE| S_IREAD | S_IWRITE, &pd, FAP_READ|FAP_WRITE, size);
if (err) {
        printf("Cannot open Pipe Error:%d\n",(int)err);
        _os_exit(err);
    }
Open:  err = _os_open ("/pipe/meine_pipe",S_IREAD | S_IWRITE,&pd);
Close: err = _os_close (pd);
```

- Programmgesteuertes Schreiben, Lesen und Statusabfragen einer named-Pipe:
  _os_read(),_os_write(), _os_gs_ready ():
- *Read: error_code = _os_read (path_id pd, void * buffer, u_int32 * count);*
  ‣count enthält beim Aufruf die Wunschanzahl und bei der Rückkehr die tatsächlich gelesene Anzahl der Bytes ‣EOF-Error erhält man erst bei einem Aufruf wenn <u>kein</u> Byte da ist.
- *Write: error_code = _os_write (path_id, void * buffer, u_int32 *count);*
  ‣count enthält beim Aufruf die Wunschanzahl und bei der Rückkehr die tatsächlich geschriebene Anzahl der Bytes

Programmierung

PIPES V

- ***Lese- Statusabfrage:***

  error_code = _os_gs_ready (path_id pd, u_int32 *size);

  - ***Der Aufruf prüft wieviel Bytes noch zu lesen da sind und gibt die Anzahl in size zurück. Gibt es <u>keine</u> so wird der Fehler EOS_NOTRDY = 246 zurückgegeben und size enthält keinen sinnvollen Wert.***

- ***Warten auf neue Zeichen mit Timeout- Abbruchmöglichkeit:***

  Idee: Man lässt sich vom Treiber ein Signal <n> schicken wenn ein Zeichen gekommen ist.

  Wichtig: Eine eigene Signalintercept-Routine muss installiert sein

  Beispiel:

```
 u_int32 time = 1000;                          /* Timeout 1000 TICS */
_os_sigmask(1);                                /*Sperre gegen Signalempfang */
_os_ss_sendsig(path_id pd, (signal_code) 350); /* Sende  mir Signal 350 wenn auf path_id pd
                                                  neue Daten da sind */
_os_sleep(&time, &sig);       /* Schlafe maximal Zeit TICS , werde geweckt entweder nach
                                 Timeout oder vorher durch Signalempfang 350 weil ein Datum da ist */
_os_ss_relea(path_id pd);    /* In jedem Fall delete den Signalsendeauftrag, egal ob
                                 Timeout oder das Signal geschickt wurde */
if ( (time!=0) || (sig==350) )
        {/* nun ist ein Zeichen da zum lesen auf path_id pd*/}
    else
       {/* Timeout!! */ }
```

Datenfluß/Inter-Prozess-Kommunikation

Shared-Memory

Gemeinsamer Speicherbereich einer oder mehrerer Tasks innerhalb eines Rechners

Realisierung:

Threads: trivial, aber kritischen Bereich nicht vergessen!!!!!!!

Prozesse:

- **Anfordern von gemeinsamem Speicher vom Betriebssystem (unterschiedliche APIs, z.B. POSIX: Name per Filesystem, SYSV: ipcs-Block)**
- **Speicheradresse des gemeinsamen Speicherbereichs kann von Task zu Task unterschiedlich sein**
- **Pointer innerhalb von Datenstrukturen relativ zum Beginn (oder Ende) des Speicherbereichs angeben**

Programmierung

Shared-Memory I OS9 (OS9 Datenmodul)

- **OS9 unterstützt im Entwicklerkernel Speicherschutzmechanismen mittels MemoryManagementUnit(MMU) ▸Realisierung von sog. shared-memory erfolgt hier mittels sog. Datenmodulen.**

- **Datenmodul ist eine Sonderform des allgemeinen Moduls ▸Modul-Type MT_DATA ; Language ML_ANY**

- **Das Datenmodul kann als Datei vorhanden sein oder es wird (meistens) von einem Prozess generiert.**

- **Ein Datenmodul ist gekennzeichnet durch einen Modulnamen und seine Größe. Die innere Strukturierung der Daten bleibt dem Nutzer überlassen. ▸Nach dem Anlegen des Moduls wird der Datenbereich gelöscht! ➔Der Erzeuger erhält Zeiger auf den Modulkopf und die Moduldaten. ▸*_os_datmod()***

- **Ein Datenmodul besitzt ebenfalls ein CRC Abschluß. ▸Soll das Datenmodul in eine Datei gespeichert werden und wieder ladbar sein ▸CRC Berechnung durchführen bevor abgespeichert wird ▸*_os_setcrc()***

- **Neben dem Erzeuger des Datenmoduls kann sich jeder, der entsprechende Zugriffsrechte (➔Modulpermissions) besitzt, Zugriff durch Anlinken verschaffen (▸Modulname dient zur Identifikation) . ▸Jeder Link erhöht den Datamodullinkcount ▸Jeder angelinkte erhält einen Zeiger auf den Modulkopf/Moduldaten. ➔*_os_link()***

- **Die synchronisierte Zugriffssteuerung auf das ggf. gemeinsame Datenmodul bleibt den Tasks überlassen. ▸OS9-Eventsteuerung, allg. Semaphore**

- **Der Erzeuger und alle angelinkten Benutzter müssen das Datenmodul wieder durch unlinken freigeben. ▸*_os_unlink()***

Programmierung

Shared-Memory II OS9 (OS9 Datenmodul)

- *Beispiel Erzeugen eines Datenmoduls:*

```
#define DATA_SIZE  1024                    /*▸Größe des Datenmodulbereichs!!*/
#define DMOD_NAME "MeinDatenModul"         /*▸Datenmodulname */
void main (int argc, char* argv[])

int * meine_daten;                  /*▸später: Zeiger auf den Beginn des Datenbereichs*/
u_int16 attr_rev, type_lang;
u_int32 perm;
error_code err;
mh_data *mod_head;

type_lang = mktypelang (MT_DATA,ML_ANY);
attr_rev = (MA_REENT <<8);
perm = MP_OWNER_READ | MP_OWNER_WRITE | MP_GROUP_READ | MP_GROUP_WRITE
|MP_WORLD_READ|MP_WORLD_WRITE ;

err = _os_datmod ( DMOD_NAME, DATA_SIZE * sizeof (int), &attr_rev,  &type_lang, perm, (void**)
    &meine_daten, &mod_head [ ,MEM_ANY ]);
```

- ▸ *Zeiger: meine_daten zeigt jetzt auf den Beginn des Datenbereichs. mod_head zeigt auf den Modulkopf*

Programmierung

Shared-Memory III OS9 (OS9 Datenmodul)

- *Anlinken auf ein bereits erzeugtes Datenmodul:*

```
#define   DMOD_NAME    "MeinDatenModul"        /*‣Datenmodulname */

void main (int argc, char* argv[])
char* d_name= DMOD_NAME;
int * meine_daten = NULL;              /*‣später: Zeiger auf den Beginn des Datenbereichs*/
u_int16 attr_rev, type_lang;
error_code    err;
mh_com       *mod_head;

type_lang = mktypelang (MT_DATA,ML_ANY);
err = _os_link ( &d_name, &mod_head, (void**) &meine_daten, &type_lang, &attr_rev ) ;
```

‣meine_daten zeigen auf den Beginn des Datenmoduls ‣Die Datenmodul-Größe muss bekannt sein!

- *Freigeben eines angelinkten Datenmodul:*

‣includes wie vorher

```
mh_com * mod_head; /*‣zeigt auf den Modulkopf!! Vorher durch link/create bekommen!!
err =_os_unlink ( mod_head );
```

Programmierung

Shared-Memory IV OS9 (OS9 Datenmodul)

- *Laden eines abgespeicherten Datenmoduls:*

```
char * mod_name = DMOD_NAME;
mh_com * mod_head;      /* Hier wird nach dem Laden der Verweis auf den Modulkopf hinterlegt*/
int* pdata ;            /* zeigt nach laden auf das erste Datenbyte des Moduls*/
u_int32 mode; u_int16 type_lang ; u_int16 attr_rev ;

type_lang = mktypelang (MT_DATA,ML_ANY);
mode = S_IREAD| S_IWRITE| S_IGREAD|S_IGWRITE|S_IOREAD|S_IOWRITE ;
error_code  _os_load( mod_name,& mod_head,(void **)pdata , mode,&type_lang,&attr_rev, MEM_ANY);
```

- *CRC-Ausrechnen*

```
mh_com * mod_head; /*‣zeigt auf den Modulkopf!! Vorher durch link/create bekommen!!
_os_setcrc ( mod_head) ;
```

- *Speichern eines Datenmoduls in eine Datei: Umweg: system ("save -f=<filename> <modulname>");*

Datenfluß/Inter-Prozess-Kommunikation

Sockets

- Kommunikation über Rechnergrenzen hinweg (aber auch lokal möglich)
- Verbindung identifiziert über (Rechnername[IP-Adresse], Port-Nummer)
- Server öffnet Port (TCP: und wartet auf Verbindungswunsch) und liest vom Socket
- Client (TCP: öffnet Verbindung und) sendet an Server

...UebersichtSocket1/2/3.pdf auf cd1.ee.hm.edu......

Programmierung

„Time triggered“ versus „Event triggered“

Busy-Loops

Bei der Busy-Loop-Lösung ist die Echtzeitsteuerung nur mit der Regelung *eines* technischen Prozesses beschäftigt

Rechenprozess *pollt* ständig den technischen Prozess, ob dieser die Anforderung stellt oder nicht

- **Einfach programmierbar**
- **meist ohne Betriebssystem**
- **sehr häufig bei Embedded Systems**

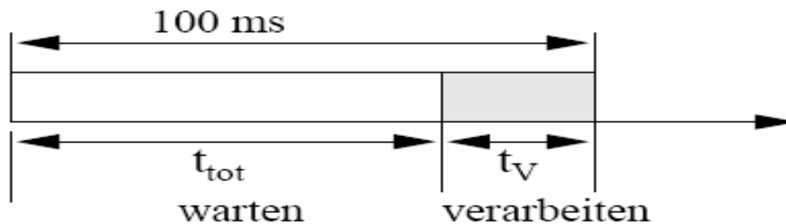
Achtung beim Pollen: `volatile`

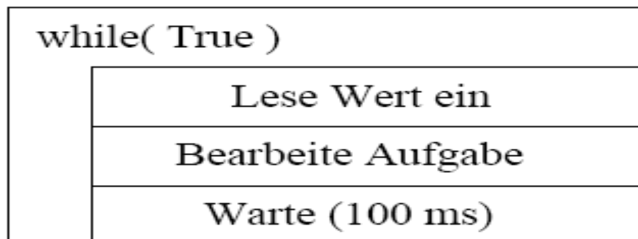
Programmierung

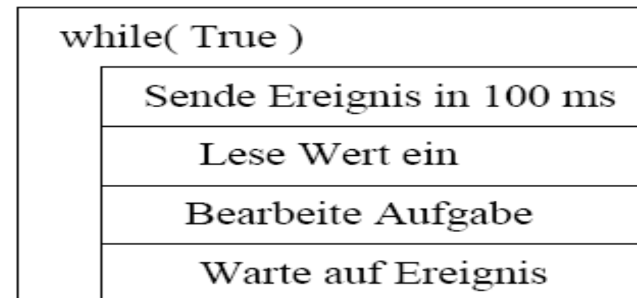
„Time triggered“ versus „Event triggered“

Time-Triggered

- **Applikation prüft zyklisch, ob bestimmte Bedingungen erfüllt sind**
- **Zeitbasis aus dem Betriebssystem**
- **Peripherie ist beim Time-Triggered-Betrieb rein passiv**
- **Abtastzeit (bzw. Prozesszeit) so kurz wählen, dass**
  - **bei der Zustandserfassung kein Zustand verloren geht (Abtasttheorem) und**
  - **eine Reaktion (2. Echtzeitbedingung) rechtzeitig erfolgen kann.**







Programmierung

„Time triggered“ versus „Event triggered“

Event Triggered

Ereignisgesteuerte Applikation reagiert auf Ereignisse oder (Software-) Interrupts, die von anderen Tasks oder vom Betriebssystem (z.B. Gerätetreiber) generiert werden.

- **Im Regelfall effizienter**
- **Determinismus: abhängig von zeitlicher Verteilung externer Events**
- **Häufig (isochroner Modus von Echtzeitbussen): Zeitbasis exakter als im Betriebssystem**

Verfahren	Charakteristikum
Busy-Loop	100%ige Auslastung, unabhängig vom Bedarfsfall
Time-Triggered	Die Auslastung ist abhängig vom gewählten Abtastintervall und unabhängig vom Bedarfsfall.
Event-Triggered	Die Auslastung ist abhängig vom Bedarfsfall.

Programmierung

Programmierstil

Stil bezüglich

- **Zerlegung eines Systems in Tasks**
- **Implementierung der einzelnen Tasks**

Zerlegung eines Systems in Tasks

- **Zeitkritische Tasks sollten möglichst wenige Systemaufrufe verwenden**
- **Sehr sorgfältiger Umgang mit “kritischen Abschnitten“, insbesondere bei Tasks mit niedriger Priorität**
  - **Entwurf**
  - **Implementierung**
- **Kommunikation zwischen Tasks ist sorgfältig zu entwerfen**
  - **Für die Kommunikation innerhalb eines Rechners: gemeinsamer Speicher (Shared Memory als kritischen Bereich, Message Queue,...)**
  - **Kommunikation von Tasks innerhalb verteilter Systeme: Sockets und darauf aufbauende Protokolle**

Programmierung

Programmierstil

Zerlegung eines Systems in Tasks

- Code und Daten müssen für kritische Realzeitprozesse immer im Hauptspeicher stehen (Stichwort: Memory-Locking).
- Zeitaufwändige Aufgaben sollten in eigene Tasks verlagert werden.
- Ein „amoklaufender“ Thread kann mehr Schaden anrichten, als ein „amok-laufender“ Rechenprozess. Daher sollten insbesondere in sicherheitskritischen Anwendungen Rechenprozesse den Threads vorgezogen werden.

Programmierung

Programmierstil

Kodierung der Tasks:

- **Rekursion vermeiden!!!**
  - **Rekursion ist ein schönes Konzept, aber nicht praxistauglich**
  - **Stack ist (insbesondere in Realzeitsystemen) eine eng beschränkte Ressource**
  - **Funktionsaufrufe sind teurer als Kontrollstrukturen**
  - **Was rekursiv programmiert werden kann, kann auch iterativ programmiert werden (Entrekursivierung)**

Programmierung

Programmierstil

Kodierung der Tasks:

- Schleifen müssen eine feste Obergrenze aufweisen
  - Bei Abbruch- bzw. Verzweigebedingungen auf Basis von variablen Werten: Zusatzbedingung für Abbruch nach einer maximalen Anzahl von Durchläufen

Gut

```
for( ; limit < CritPoint(v); limit=lnext(limit) ) {
...
}
```

besser

```
for( n=0; (limit < CritPoint(v)) && (n < 5000);
     limit=lnext(limit), n++) {
...
}
if( n == 5000 ) {
... // Fehlerbehandlung
```

Programmierung

Programmierstil

Datenstrukturen

- **statisch definieren oder**
- **zur Initialisierungszeit allozieren**
- **<u>nicht erst zur Laufzeit Speicher allozieren</u>**
  - **undeterministisches Verhalten der Speicherverwaltung**
  - **Risiko, dass kein Speicher mehr vorhanden**

Bibliotheksfunktionen: darauf achten, dass diese

- **realzeitfähig und**
- **Ggf. multitaskingfähig (reentrant, thread-safe)**

Rückgabewerte von Funktionen (insbesondere Return-Codes)

- **grundsätzlich in ihrem kompletten Definitionsbereich auswerten!!!!!!!!!!!!!!!!!!**

Zusammenfassung

- **Programmier-Methoden bei Echtzeitsystemen**
  - **SPS-Programmierung**
  - **Zustandsautomaten (*state machines*)**
  - **Programmiersprachen**
- **Multitasking/Kontrollfluss**
  - **Kritischer Bereich**
  - **Semaphore/Mutex**
  - **Prioritätsinversion**