Echtzeitsysteme

7. Programmierung planbarer Systeme

Prof. Dr. Roland Dietrich

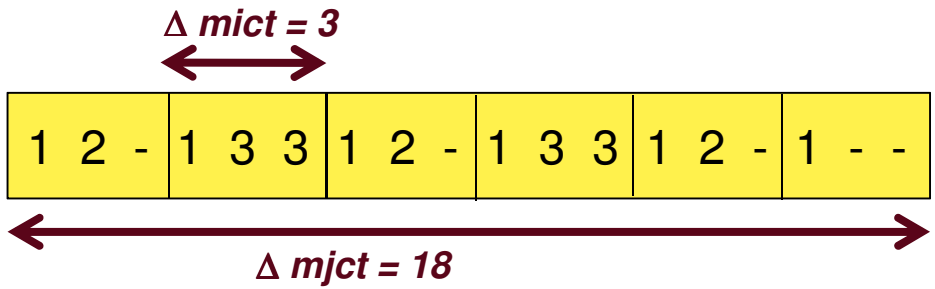
Programmierung zyklischer Pläne

- Erforderliche Sprach-Unterstützung
  - Zugriff auf Zeit oder Interrupts
- Beispiel:

Taskmenge:

T	Δe	Δt
1	1	3
2	1	6
3	2	9

Zyklischer Plan (6 Runden):



Programm mit Zeitrechnung:

```
dmict = 3; rnd = 6;
cnt =0; t = gettime();
while (TRUE){
  wait_until(t);
  switch (cnt) {
    case 0: T1;T2;
    case 1: T1;T3;
    case 2: T1;T2;
    case 3: T1;T3;
    case 4: T1;T2;
    case 5: T1;
  }
  cnt =(cnt+1) % rnd;
  t=t+dmict;
  if(gettime()>t)
    handle time overrun
} // end while
```

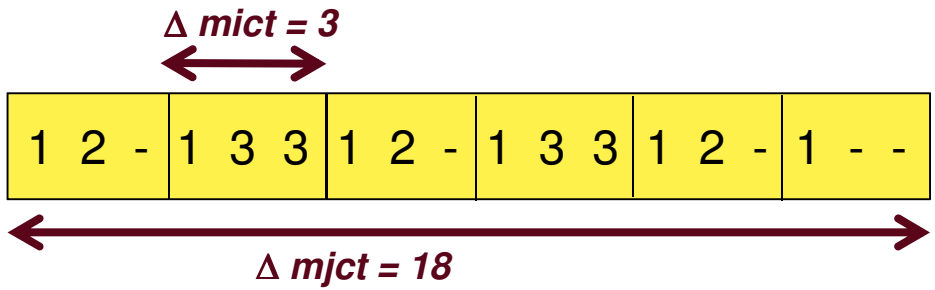
Programmierung zyklischer Pläne

- Beispiel: Ausführung eines zyklischen Plans mit Interrupts
  - Zu jedem Beginn eines inneren Zyklus wird ein Interrupt ausgelöst
    - Programmierbar z.B. über Timer

Taskmenge:

T	Δe	Δt
1	1	3
2	1	6
3	2	9

Zyklischer Plan:



Programm mit Interrupts:

```
loop - Major Cycle
  wait_for_Interrupt;
  T1; T2; -- Minor cycle 1
  wait_for_Interrupt;
  T1; T3; -- Minor cycle 2
  wait_for_Interrupt;
  T1; T2; -- Minor cycle 3
  wait_for_Interrupt;
  T1; T3; -- Minor cycle 4
  wait_for_Interrupt;
  T1; T2; -- Minor cycle 5
  wait_for_Interrupt;
  T1;     -- Minor cycle 6
end loop;
```

Programmierung zyklischer Pläne

- Fehlerbehandlung
  - Programm mit Zeitrechnung (vgl. S. 7-2)
    - Jeder Schleifendurchgang entspricht einem äußeren Zyklus.
    - Am Ende des Schleifenrumpfs kann die Zeiteinhaltung geprüft (und behandelt) werden
  - Programm mit Interrupts
    - Ein **Flag** wird am Ende eines inneren Zyklus auf ***1*** gesetzt
    - Die Interrupt-Behandlungsroutine prüft dieses Flag und setzt es auf ***0***

```
inTime = 1;
loop - Major Cycle
  wait_for_Interrupt;
  T1; T2; -- Minor cycle 1
  inTime = 1;
  wait_for_Interrupt;
  T1; T3; -- Minor cycle 2
  inTime = 1;
  . . .
end loop;
```

Interrupt-Behandlungsroutine:

```
void HandleInterrupt() {
  if (inTime)
    -- minor cycle ended
    -- in time
    inTime = 0;
  else
    handle time overrun;
  end if;
}
```

Programmierung prioritätsbasierter Systeme

- Ausgangspunkt: eine als FPS-planbar nachgewiesene Taskmenge
  → Planbarkeitstests!
- Erforderliche Sprach-/Laufzeitunterstützung
  - Möglichkeit, Prioritäten für Tasks zu definieren
  - Unterstützung von Verdrängung (*preemtive multitasking*)
    - sofortiger Taskwechsel: eine Task *T* wird rechnend und verdrängt eine Task *T'*
      - sobald *T* rechenbereit ist
      - und *T* eine höhere Priorität hat als *T'*
  - Unterstützung von Prioritätsvererbung
    - noch besser: Prioritätsobergrenzen (*priority ceiling protocol*)
- Praxis
  - FPS wird unterstützt in Ada, C/Real-Time POSIX und den meisten kommerziellen Echtzeitbetriebssystemen

Ada und FPS

- Echtzeitmodell ist definiert im ***ADA Real-Time Annex***
  - Statische Basisprioritäten (***base priorities***)
  - Dynamische Basisprioritäten (***dynamic priorities***)
  - Gegenseitiger Ausschluss in geschützten Objekten durch Prioritätsobergrenzen (***priority ceiling locking***)
  - Laufzeitprioritäten (***active priorities***)
  - Verschiedene Strategien für die Prozessorzuteilung (***dispatching policies***)
  - Spezielle Profile, z.B. *Ravenscar Profile*
    - Definiert eine Teilmenge der Ada-Tasking-Möglichkeiten für besonders sicherheitskritische Systeme
- Definition der Prozessorzuteilungsstrategie durch ein **pragma** (Compiler-Direktive), z.B.

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities)
```

  - Tasks mit einer Priorität p, die rechenbereit werden, werden in einer Bereit-Warteschlange (ready-queue) für p verwaltet

- **Datentypen für Prioritäten**
  - Typ-Deklarationen im Package **System**

```
subtype Any_Priority is Integer
        range Implementation-Defined;
subtype Priority is Any_Priority range
        Any_Priority'First .. Implementation-Defined;
subtype Interrupt_Priority is Any_Priority range
        Priority'Last + 1 .. Any_Priority'Last;
Default_Priority : constant Priority :=
        (Priority'First + Priority'Last)/2;
```

  - Eine Ada-Implementierung muss mindestens 30 Prioritäten im Typ **Prority** und mindestens eine davon verschiedene Interrupt-Priorität im Typ **Interrupt_Priority** definieren.

Ada und FPS

- Statische **Basisprioritäten** zuteilen
  - Bei Task-Definitionen

```
task Controller is
  pragma Priority(10);
  ...
end Controller;
```

  - Bei Task-Typdefinitionen: Jede Instanz des Typs kann eine andere Priorität haben

```
task type Servers(Pri : System.Priority) is
  -- each instance of the task can have a
  -- different priority
  entry Service1(...);
  entry Service2(...);
  pragma Priority(Pri);
end Servers;
```

Ada und FPS

- Statische Basisprioritäten zuteilen
  - Priorität des Hauptprogramms
    - Kann definiert werden durch das `Priority`-Pragma
  - Default-Werte für Prioritäten
    - Hauptprogramm: `Default_Priority` (vgl. S. 7-7)
    - Andere Tasks: Die Priorität der Task, die sie erzeugt hat

- Statische Basisprioritäten für geschützte Objekte als Interrupt-Handler (vgl. S. 5-39f)

```
pragma Interrupt_Priority(Expression);
```

  oder

```
pragma Interrupt_Priority;
```

  - *Expression* kann zu beliebiger Priorität evaluieren (`AnyPriority`)
    - D.h. man kann auch niedrige Prioritäten für Interrupts definieren
  - Default (**pragma** ohne Parameter): `Any_Priority'Last`

- **Gegenseitiger Ausschluss mit Prioritäts-Obergrenzen (*priority ceiling locking*)**
  - Einem geschützten Objekt wird eine **Prioritätsobergrenze** (***ceiling priority***) zugewiesen
    - größer oder gleich der höchsten Priorität einer Task, die das geschützte Objekt benutzt (vgl. 6-35, p_{max})
  - Wenn eine Task eine geschützte Operation aufruft, wird ihre Priorität auf die Prioritätsobergrenze des geschützten Objekts gesetzt
    - **Aktive Priorität**, (vgl. 6-35, p_{run})
  - Folge
    - Wenn eine Task eine geschütztes Objekt betreten möchte, kann dieses nicht schon belegt sein

    → Gegenseitiger Ausschluss ist garantiert

Ada und FPS

- Gegenseitiger Ausschluss mit Prioritätsobergrenzen (*priority ceiling locking*)
  - Zuweisung der Prioritätsobergrenze durch `pragma(Priority)` im geschützten Objekt
    - D.h. die Prioritätsobergrenze wird nicht dynamisch ermittelt, sie wird vom Programmierer festgelegt
    - Die Ausnahme `Program_Error` wird (zur Laufzeit!) ausgelöst, falls die aktive Priorität einer Task vor Betreten eines geschützten Objekts größer ist als dessen Obergrenze
    - Intensives Testen oder genaue statische Analyse erforderlich!
  - Default Obergrenze: `Priority'Last`
- Effektive Implementierung
  - Wenn eine Task eine geschützte Operation ausführt, die Blockierungen anderer Tasks aufhebt, kann sie auch gleich den "freigegeben" Code der anderen Tasks ausführen

  → Weniger Task-Wechsel erforderlich

Ada und FPS

- Gegenseitiger Ausschluss mit Prioritäts-Obergrenzen (*priority ceiling locking*) – Beispiel ([Burns & Wellings 2005, Kap. 12.3])

```
protected Gate_Control is
   pragma Priority(28);
   entry Stop_And_Close;
   procedure Open;
private
   Gate : Boolean := False;
end Gate_Control;
```

```
protected body Gate_Control is
   entry Stop_And_Close
            when Gate is
   begin
      Gate := False;
   end;
   procedure Open is
   begin
      Gate := True;
   end;
end Gate_Control;
```

Ada und FPS

- Gegenseitiger Ausschluss mit Prioritäts-Obergrenzen (*priority ceiling locking*) – Beispiel ([Burns & Wellings 2005, Kap. 12.3])
  - Beispiel-Ablauf (nach ICPP): Tasks **T** (Priorität=20) und **S** (Priorität=27)
    - T ruft `Stop_And_Close` → wird blockiert (`Gate` ist `false`)
    - Später: S ruft `Open`
    - Task S führt `Open` aus
    - T führt `Stop_And_Close` aus (`Gate` ist `true`) (Taskwechsel!)
      - Aktive Priorität von T ist 28 > 27!
    - S wird fortgesetzt nach dem Aufruf von Open (Taskwechsel!)
  - Effizienterer Ablauf:
    - T ruft `Stop_And_Close` → wird blockiert (`Gate` ist `false`)
    - Später: S ruft `Open`
    - Task S führt `Open` aus
    - **S (!)** führt `Stop_And_Close` für T aus (`Gate` ist `true`) (**kein** Taskwechsel!)
    - S wird fortgesetzt nach dem Aufruf von Open (**kein** Taskwechsel!)

- **Prioritäten zur Laufzeit** (***active priorities***)
  - Die **Laufzeitpriorität** (***active priority***) einer Task ist das Maximum zwischen der eigenen statischen Priorität und der **geerbten Priorität**
  - *Wann kann eine Task eine Laufzeitpriorität erben, die höher ist als die eigene statische?*
    - Eine Task, die ein geschütztes Objekt betritt, erbt die Prioritätsobergrenze des geschützten Objekts
    - Wenn eine Task erzeugt wird, erbt sie die aktive Priorität ihrer Eltern-Task
      - Beachte: die erzeugende Task muss warten, bis ihre Kinder beendet sind!
    - Während eines Rendezvous: Die Task, die ein `accept`-Anweisung ausführt, erbt die Laufzeitpriorität der den Eintrittspunkt aufrufenden Task

Ada und FPS

- **Taskwechsel (*dispatching*)**
  - Die Reihenfolge der Taskausführung ergibt sich aus der Laufzeitpriorität
  - Taskwechsel-Strategie kann definiert werden über das Pragma `Task_Dispatching_Policy`
    - Default: FPS mit verdrängen
    - Seit Ada95 ist eine Taskwechsel-Strategie definiert: `FIFO_Within_Priority`
      - Wenn eine Task ausführbereit wird (z.B. am Ende einer Blockade), wird sie am Ende der Wartschlage für ihre Priorität eingereiht
      - Wenn eine Task verdrängt wird (weil eine Task mit höherer aktiver Priorität rechenbereit ist), wird sie am Anfang der Warteschlange für ihrer Priorität eingereiht
    - Ada2005 definiert weitere Taskwechsel-Strategien, z.B. `Non_Preemtive_FIFO_Within_Priority`
      - Tasks werden nicht verdrängt, sondern werden vor dem Taskwechsel zu ende ausgeführt

Dynamische Basisprioritäten in Ada

- Basisprioritäten sind normalerweise statisch
  - sind über die gesamte Laufzeit einer Task fest
- *Warum sollte sich die Priorität zur Laufzeit ändern können?*
  - Implementierung von Betriebsarten
    - Eine Task kann in einer bestimmten Betriebsart (z.B. "Notbetrieb") eine höhere/niedrigere Priorität haben als in einer anderen (z.B. "Normalbetrieb")
  - Anwendungsspezifische Prioritäten
- In Ada:
  - Tasks können ihre Basispriorität (eigentlich *statische* Priorität) zur Laufzeit ändern
  - Geschützte Objekte können ihre Prioritätsobergrenzen zur Laufzeit ändern

Dynamische Basisprioritäten in Ada

- Das Package `Ada.Dynamic_Priorities`

```
with Ada.Task_Identification; use Ada;
package Ada.Dynamic_Priorities is
   procedure Set_Priority(Priority : System.Any_Priority;
      T : Task_Identification.Task_Id :=
      Task_Identification.Current_Task);
   function Get_Priority(T : Task_Identification.Task_Id
       := Task_Identification.Current_Task)
       return System.Any_Priority;
       -- raise Tasking_Error if task has terminated
   -- Both raise Program_Error if a Null_Task_Id is passed
private
   -- not specified by the language
end Ada.Dynamic_Priorities;
```

Programmierung Fristenbasierter Systeme

- Ausgangspunkt: eine als EDF-planbar nachgewiesene Taskmenge
  → Planbarkeitstests!
- Erforderliche Sprach-/Laufzeitunterstützung
  - Die Möglichkeit, Deadlines für Tasks zu definieren
  - Unterstützung von Verdrängung basierend auf Deadlines
    - sofortiger Taskwechsel: eine Task *T* wird rechnend und verdrängt eine Task *T'*
      - sobald *T* rechenbereit ist
      - und *T* eine nähere Deadline hat als *T'*
  - Eine Möglichkeit, gemeinsame Ressourcen zu nutzten, die kompatibel ist mit EDF
    - analog zu Prioritätsobergrenzen
- Praxis:
  - Wird in aktuellen Programmiersprachen/Betriebssystemen meist nicht unterstütz – außer in Ada

Ada und EDF

- **Spezifikation von Fristen** (*deadlines*)
  - Das Package Ada.Dispatching.EDF

```
package Ada.Dispatching.EDF is
   subtype Deadline is Ada.Real_Time.Time;
   Default_Deadline : constant Deadline :=
                        Ada.Real_Time.Time_Last;
   procedure Set_Deadline(D : in Deadline;
                          T : in Ada.Task_Identification.Task_ID :=
                          Ada.Task_Identification.Current_Task);
   procedure Delay_Until_And_Set_Deadline(
                          Delay_Until_Time : in Ada.Real_Time.Time;
                          TS : in Ada.Real_Time.Time_Span);
   function Get_Deadline(
      T : inAda.Task_Identification.Task_ID
          := Ada.Task_Identification.Current_Task)
      return Deadline;
end Ada.Dispatching.EDF;
```

Ada und EDF

- **Beispiel:** Schema für die Programmierung einer periodischen Task (Deadline = Periodenende):

```
task body Periodic_Task is
    Interval : Time_Span := Milliseconds(30);
    -- define the period of the task, 30ms in this example
    -- relative deadline equal to period
    Next : Time;
  begin
    Next := Clock;  -- start time
    Set_Deadline(Clock+Interval);
    loop
      -- undertake the work of the task
      Next := Next + Interval;
      Delay_Until_And_Set_Deadline(Next,Interval);
    end loop;
  end Periodic_Task;
```

Ada und EDF

- Spezifikation von Fristen – **Initiale Fristen**
  - Problem:
    - eine Task kann mit `Set_Deadline` erst eine Frist erhalten, wenn sie gestartet ist
    - Die Default-Frist ist `Default_Deadline` (vgl. S. 7-19), das ist in "sehr ferner Zukunft"
    - Alle Tasks mit spezifizierter Deadline werden zunächst vorgezogen
  - Lösung:
    - **`pragma`** `Realtive_Deadline(`*`Realative_Deadline_Expression`*`)`
      - *`Realative_Deadline_Expression`* ist vom Typ `Ada.Real_Time.Time_Span`
      - Zwischen Task-Erzeugung und Task-Aktivierung wird die initiale Deadline festgelegt zu

        `Ada.Real_Time.Clock +` *`Realative_Deadline_Expression`*
  - Beispiel (vgl. S. 7-20) : Statt des `Set_Deadline`-Aufrufs

```
task Periodic_Task is
  pragma Relative_Deadline(Milliseconds(30));
end Periodic_Task;
```

Ada und EDF

- Entdeckung und Behandlung von **Fristverletzungen**
  - Beispiel:

```
loop
  select
    delay until Ada.Dispatching.EDF.Get_Deadline;
    -- action to be taken when deadline missed
    then abort
    -- code
  end select;
end loop;
```

Ada und EDF

- Taskwechsel-Strategie
  - EDF-Planung wird festgelegt durch das Pragma

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
```

  - Strategie
    - Die Warteschlangen werden nach wie vor nach (aktiven) Prioritäten geführt
    - Die Ordnung in den Warteschlangen ist nicht FIFO, sondern nach "earliest deadlines" (kürzeste absolute Fristen zuerst)
    - Die Regeln zur Berechnung der aktiven Prioritäten sind anders (s.u.)
    - Regeln für den Fall "keine Kommunikation"
      - Alle in den Task gesetzten Basisprioritäten werden ignoriert
      - Alle Tasks haben eine (aktive) Priorität: `System.Priority'First`

Ada und EDF

- Task-Wechselstrategie mit Kommunikation: ***Baker's Algorithm***
  - Parameter:
    - Die Deadlines der Tasks repräsentieren Dringlichkeit (→ EDF)
    - Die Tasks erhalten eine Priorität (Basispriorität, wie bei FPS)
      - z.B. festgelegt nach monotonen Fristen
    - Die geschützten Objekte erhalten eine Priorität (Prioritätsobergrenze, wie bei FPS)
  - Regeln:
    - Wenn eine Task ein geschütztes Objekt betritt, wird sie mit dessen Prioritätsobergrenze als aktive Priorität ausgeführt
    - Wenn eine Task S in einem geschützten Objekt rechnet und eine Task T wird rechenbereit, dann wird S durch T verdrängt, wenn
      - T eine kürzere absolute Deadline als S hat
      - T eine höhere aktive Priorität als S hat
    - Ansonsten wird T in die Warteschlange für `Priority'First` eingereiht

- FPS- Beispiel:

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities)

task Example is
  pragma Priority(5);
end Example;

task body Example is
  Next_Release : Ada.Real_Time.Time;
  Period : Ada.Real_Time.Time_Span
    := Ada.Real_Time.Milliseconds(10);
begin
  Next_Release := Ada.Real_Time.Clock;
  loop
    -- code
    Next_Release := Next_Release + Period;
    delay until Next_Release;
  end loop;
end Example;
```

- EDF- Beispiel:

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities)

task Example is
  pragma Priority(5);
  pragma Relative_Deadline(10); -- gives an initial relative
end Example;                    -- deadline of 10 milliseconds

task body Example is
  Next_Release: Ada.Real_Time.Time;
  Period : Ada.Real_Time.Time_Span
    := Ada.Real_Time.Milliseconds(10);
begin
  Next_Release := Ada.Real_Time.Clock;
  loop
    -- code
    Next_Release := Next_Release + Period;
    Delay_Until_and_Set_Deadline(Next_Release, Period);
  end loop;
end Example;
```

- **Das Ravenscar-Profil**
  - Ada ermöglicht die Definition von sogenannten **Profilen**
    - Teilmenge von Ada, die von einer **konformen Implementierung** beachtet werden muss
  - Ziel des **Ravenscar**-Profils
    - Effiziente, gut planbare und vorhersagbare Echtzeitsysteme entwicklen
  - Deklaration im Programm:

    pragma Profile(Ravenscar);

    → Der Compiler prüft die Einhaltung der Einschränkungen
  - Das Ravenscar-Profil schränkt nur die Möglichkeiten bezüglich Tasking ein
    - keine Einschränkungen für den sequentiellen Teil der Sprache
  - Kompatible Programme erschließen sich gut den behandelten Planungs- und Analyseverfahren

- Das Ravenscar-Profil
  - Was z.B. erlaubt ist:
    - Task-Typen und Objekte nur auf Bibliotheks-Ebene
    - Protected Types und protected objects nur auf Bibliotheksebene
    - **Ein** Eintrittspunkt pro geschütztem Objekt, höchstens **eine** wartende Task pro geschütztem Objekt
    - Guards in Eintrittspunkten sind eingeschränkt auf **eine** boole'sche Variable
    - delay until – Anweisung
    - Task-Wechselstrategien: `FIFO_Within_Priority` mit C`eiling_Locking`
    - Geschützte Prozeduren als Interrupt-Handler
  - Was z.B. nicht erlaubt ist:
    - Select-Anweisung (d.h. kein Rendezvous!)
    - Delay-Anweisung
    - Hierarchisch geschachtelte Tasks
    - Terminierende Tasks
      - alle Tasks existieren für die gesamte Laufzeit des Systems

- Das Ravenscar-Profil
  - Definiert über Pragmas:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;

pragma Restrictions(
  No_Abort_Statements,
  No_Dynamic_Attachment,
  No_Dynamic_Priorities,
  No_Implicit_Heap_Allocations,
  No_Local_Protected_Objects,
  No_Local_Timing_Events,
  No_Protected_Type_Allocators,
  No_Relative_Delay,
  No_Requeue_Statements,
  No_Select_Statements,
  No_Specific_Termination_Handlers,
  No_Task_Allocators,
  No_Task_Hierarchy,
  No_Task_Termination,
  Simple_Barriers,
  Max_Entry_Queue_Length => 1,
  Max_Protected_Entries => 1,
  Max_Task_Entries => 0,
  No_Dependence =>
  Ada.Asynchronous_Task_Control,
  No_Dependence => Ada.Calendar,
  No_Dependence =>
      Ada.Execution_Time.Group_Budget,
  No_Dependence =>
      Ada.Execution_Time.Timers,
  No_Dependence =>
      Ada.Task_Attributes);
```

Scheduling in C/Real-Time POSIX

- POSIX unterstützt FPS, Prioritätsvererbung und Proritätsobergrenzen (*priority ceiling protocol*)
- Prioritäten können zur Laufzeit gesetzt werden
- **POSIX-Profile** und unterstützte Merkmale (u.a.)
  - PSE51: minimales Echtzeit-Profil
    - Threads, FPS, Mutexe mit Prioritätsvererbung, Bedingungsvariablen, Semaphore, Singale, einfache Geräte-E/A – Analog Ravenscar
  - PSE52: Erweiterung von PSE51 für mehrere Prozessoren
    - Multiprozessoren, Dateisystem, Nachrichtenwarteschlangen
  - PSE53: Erweiterungen von PSE52 für Einzel- oder Mehrprozessorsysteme
    - Mehrere Prozesse mit mehreren Threads, Asynchrone E/A
  - PSE54: *Multipourpose real-time system profile*
    - Mischung von Echtzeit- und Nicht-Echtzeitsystemen auf einem oder mehreren Prozessoren
    - Netzweke

- **Taskwechsel-Strategien**
  - **FIFO:**
    - Wenn ein Prozess/Thread von einem anderen mit höherer Priorität verdrängt wird, wird er am Kopf der Warteschlage für seine Priorität eingereiht
  - **Round-Robin:**
    - Die Prozess-Laufzeit wird in Quanten eingeteilt
    - Wenn ein Prozess/Thread von einem anderen mit höherer Priorität verdrängt wird, wird er am Kopf der Warteschlange für seine Priorität eingereiht
    - Wenn sein Quantum abgelaufen ist, wird ans Ende der Warteschlange für seine Priorität eingereiht
  - Weitere Strategien: ***Sporadic Server***, **OTHER** (definiert durch die spezielle POSIX-Implementierung)

- **POSIX Prioritäten**
  - Jede Strategie hat eine minimale Anzahl von Prioritäten (FIFO, RR: 32)
  - Die Strategie/Prioritäten können für Prozesse und Tasks gesetzt werden
  - Threads können erzeugt werden mit der Option "System-Wettbewerb" (system contention) oder "Prozess-Wettbewerb" (*process contention*)
    - ***system contention***: Threads stehen im Wettbewerb um die Systemressourcen mit allen anderen Threads des Systems (auch die in anderen Prozessen)
    - ***process contention***: Threads stehen im Wettbewerb um System-Ressourcen mit anderen Threads (erzeugt mit der Option *process contentio*n) des Eltern-Prozesses.
      - Es ist nicht spezifiziert, wie der Wettbewerb mit Threads aus anderen Prozessen oder Threads im System-Wettbewerb gelöst wird
  - Eine POSIX-Implementierung muss definieren, welche Optionen (eine oder beide) sie unterstützt
- **API**: siehe [Beispiele zu Kapitel 7]

Literatur

[Burns & Wellings 2009] Alan Burns, Andy Wellings: *Real-Time Systems and Programming Languages. Ada, Real-Time Java and C/Real-Time POSIX*. Addison Wesley, 2009.