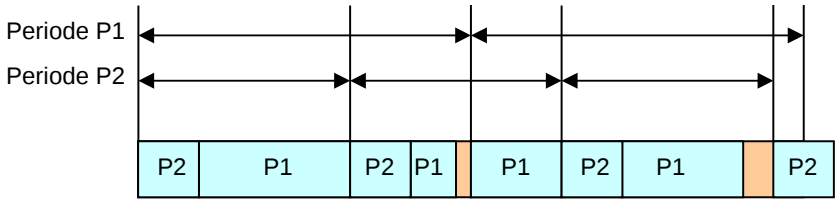
UNIVERSITÄT KOBLENZ · LANDAU

Softwarekomponete zur Abbildung von Echtzeitprozessen auf POSIX-Threads

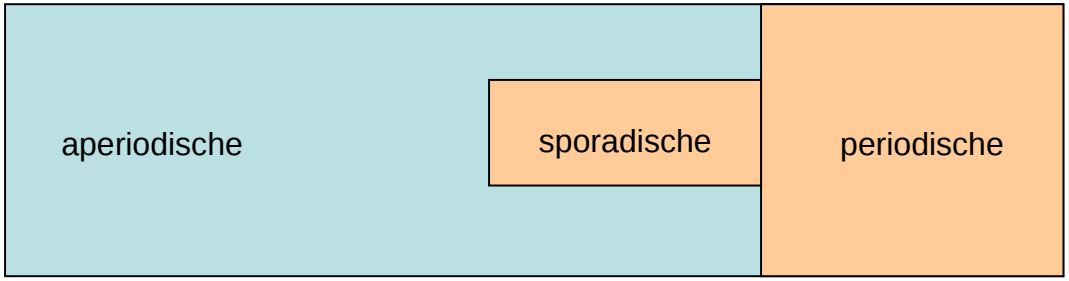
Andreas Blüm, Nils Breest, Volkmar Kobelt, Manuel Spies, Andreas Stahlhofen und Dieter Zöbel,
Institut für Softwaretechnik, Fachbereich Informatik, Universität Koblenz-Landau



ist

Planung bei Echtzeitsystemen

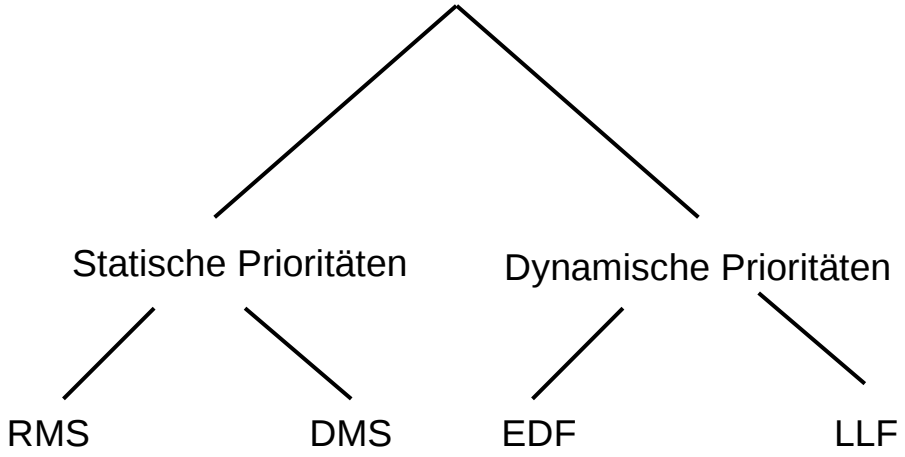
- Aspekt Unterbrechbarkeit: unterbrechbare und nicht unterbrechbare
- Aspekt Wiederholung: aperiodische, periodische und sporadische



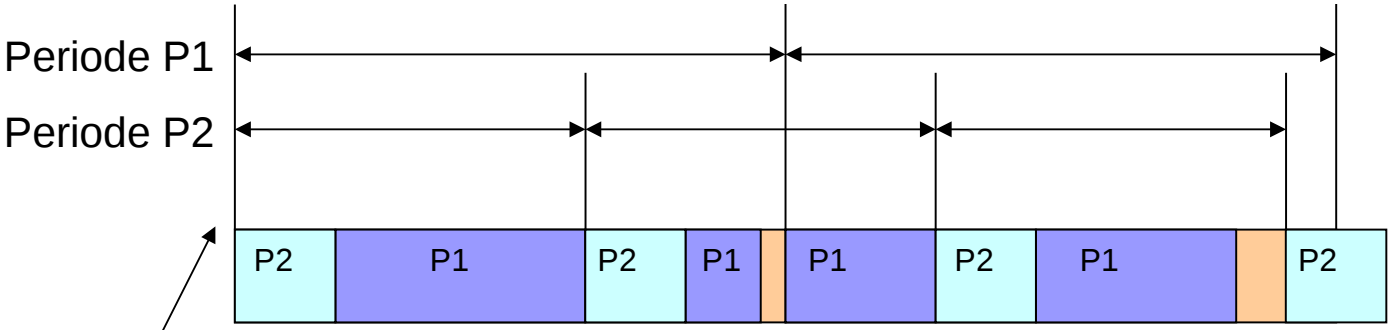
Scheduling für periodische unterbrechbare Prozesse

Prozessbeschreibungen durch Angabe der Periode Δp und der Ausführungszeit Δe

P	Δp	Δe
1	21	10
2	12	5



Beispiel: Zwei Prozesse mit Rate Monotonic Scheduling (RMS) auf einem Prozessor



Prioritätsbasiertes Scheduling

Alle Prozesse starten zum selben Zeitpunkt

P	Δp	Δe
1	21	13
2	12	3

Standardverfahren zur Überprüfung der Einhaltung aller Fristen

Beispiel: Umsetzung von RMS in die Programmiersprache PEARL (Process and Experiment Automation Realtime Language) nach DIN 66253-2 (PEARL 90)

- 1. Schritt:
  Deklaration der Prozesse (Tasks, Threads) mit fester Priorität
- 2. Schritt:
  Aktiviere die unterbrechbaren Prozesse ab einem Zeitpunkt mit einer eigenen Periode

Analoges gilt für die Programmiersprache ADA

Schema für Schritt 1: Deklaration des Hauptprogramms und der einzelnen Prozesse

Schema:

```
MODULE
  :
S: TASK MAIN;
    !task body
   END;
  :
pi: TASK PRIO i;
    !task body
    END;
  :
MODEND;
```

Schema für Schritt 2: in `MAIN` steht für jeden Prozess Pi die Anweisung:

```
AT t ALL d ACTIVATE pi;
```

Beispiel:

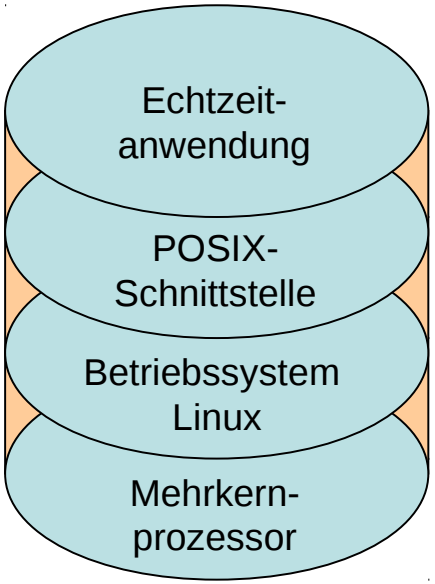
```
  :
DCL mittag CLOCK;
mittag=12:00:00;
DCL umlauf DURATION;
umlauf=0.04 SEC;
  :
  :
AT mittag ALL umlauf ACTIVATE pi;
  :
```

Frage: Wie lässt sich dies ähnlich einfach auf POSIX abbilden?

Gegenüberstellung

PEARL	ADA	POSIX
TASK	TASK	thread
PRIO	PRIO	Zuordnung einer BS-spezifischen Periode
AT	ACCEPT	Startzeitpunkt durch …
ALL	ACCEPT	Periode …

Zielsystem



Realtime Prozesskomponente

Mit POSIX-Threads

Realtime Prozesskomponente

- POSIX Library
- C++
- Linux
- Echtzeitprozesse

Features

- Prioritäten
- Periodizitäten
- Startzeitpunkt
- → Threads
  - CPU-Affinität
  - Joins
  - Termierung

POSIX – Prioritäten

- Linux: Wert von 0 – 99
  - 0: Standard Scheduler
  - 1-99: Durch POSIX spezifizierte Scheduler entsprechend interpretiert
    - 1: Minimum
    - 99: Maximum
- Andere OS: Intervall ggf. abweichend

POSIX - Scheduler

- FIFO – Scheduler
  - Ein arbeitender Thread wird nur durch höher priorisierte Threads unterbrochen
- RR – Scheduler
  - Zeitscheiben Prinzip zw. gleich-priorisierten Threads
- Standard Scheduler

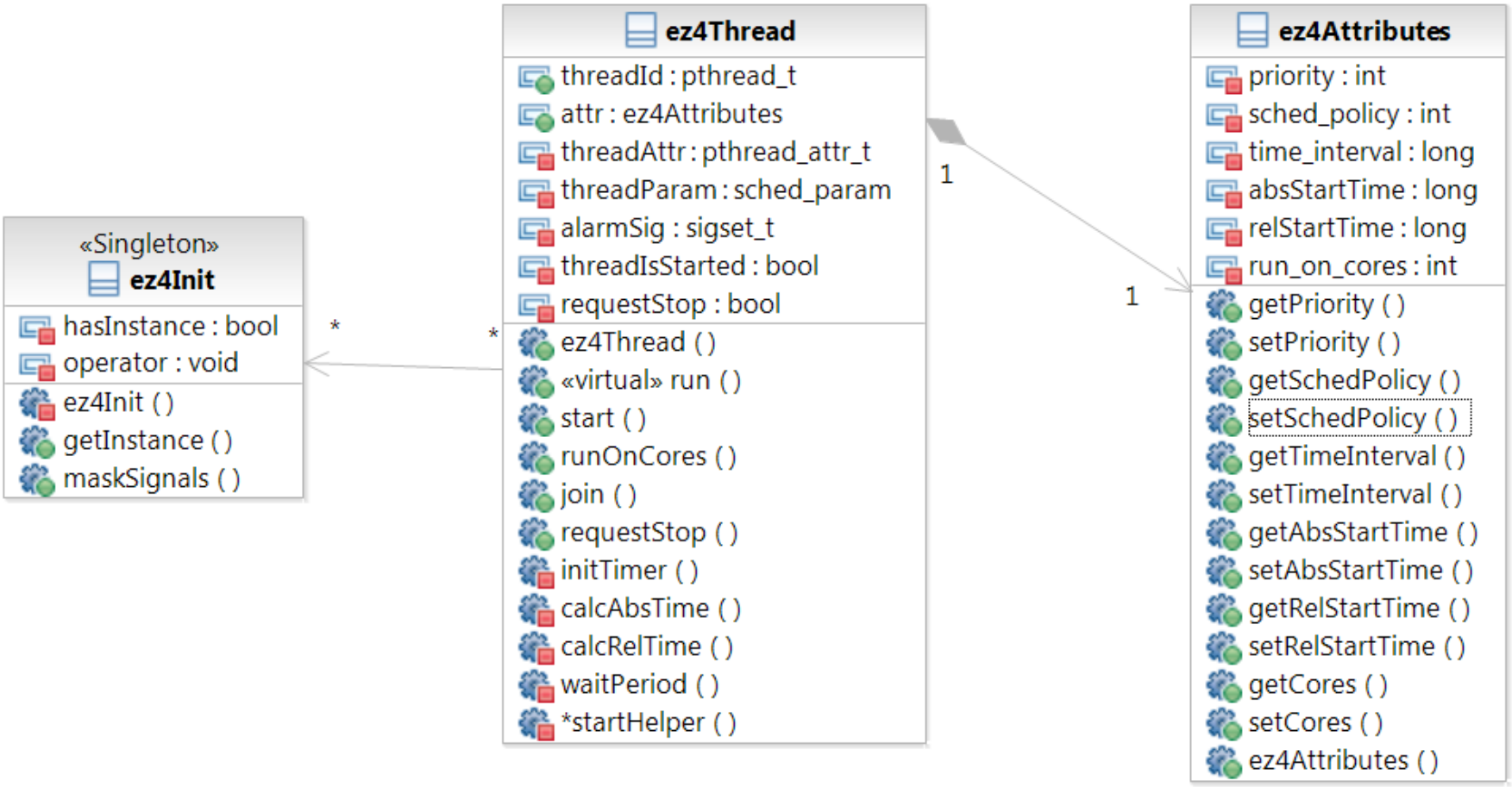
POSIX – Perioden

- Startzeitpunkt
  - Absolut: Unix Timestamp (Sekunden)
  - Relativ: In Bezug auf den “Startzeitpunkt” des Threads (ms)
- Periode in (µs)
- Verwendung des POSIX-Echtzeittimers

Timer Initialisierung

- Einmalig nach Programmstart
- Reset aller Timer für den Prozess
- Für Echtzeittimer Vorraussetzung

Aufbau


«Singleton»
ez4Init
hasInstance : bool
operator : void
ez4Init ( )
getInstance ( )
maskSignals ( )
*
*
ez4Thread
threadId : pthread_t
attr : ez4Attributes
threadAttr : pthread_attr_t
threadParam : sched_param
alarmSig : sigset_t
threadIsStarted : bool
requestStop : bool
ez4Thread ( )
«virtual» run ( )
start ( )
runOnCores ( )
join ( )
requestStop ( )
initTimer ( )
calcAbsTime ( )
calcRelTime ( )
waitPeriod ( )
*startHelper ( )
1
1
ez4Attributes
priority : int
sched_policy : int
time_interval : long
absStartTime : long
relStartTime : long
run_on_cores : int
getPriority ( )
setPriority ( )
getSchedPolicy ( )
setSchedPolicy ( )
getTimeInterval ( )
setTimeInterval ( )
getAbsStartTime ( )
setAbsStartTime ( )
getRelStartTime ( )
setRelStartTime ( )
getCores ( )
setCores ( )
ez4Attributes ( )

Anwenderfreundlich

- Getter / Setter nur auf 'int' oder 'long'
- Robustheit (Logische Fehleingaben werden abgefangen, dass keine Laufzeitfehler auftreten.)
  - In Zukunft Warnungen zur Compiletime

Anwendung

- Ableitung der ez4Thread-Klasse
- Implementation der AL in spezialisierter Klasse innerhalb einer “run()”-Funktion
- API-Funktionen werden von Metaklasse geerbt
- Ausführung mittels “start()”-Funktion, die den POSIX-Thread mit den gewählten Eigenschaften erstellt und ausführt

C vs. C++

- Anwendungsprogrammierung
- POSIX nur für C spezifiziert
- pthread_create → Kein call von Memberfunktionen “this->run()”
- Bypass über statische Memberfunktion mit Zielinstanz als Argument

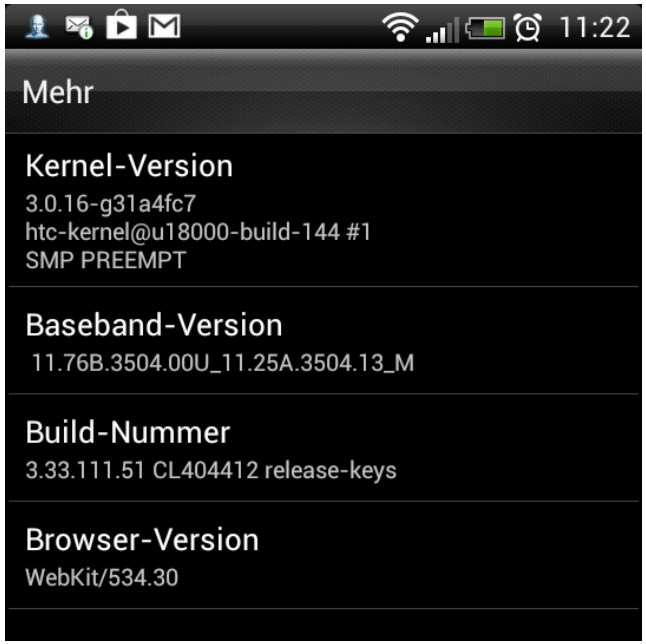
C vs. C++

```
pthread_create(ThreadID,Attributes,startHelper,this);

static void *startHelper(void* instance)
{
    ez4Thread* inst = (ez4Thread*)instance;
    inst->runOnCores(inst->attr.getCores());

    if (inst->attr.getTimeInterval() > 0)
    {
        While (1) {
            inst->waitPeriod();
            inst->run();
            if (inst->requestedStop)
            {
                inst->requestedStop = false;
                break;
            }
        }
    }
    ....
    ....
    ....
    pthread_exit(EXIT_SUCCESS);
    return 0;
}
```

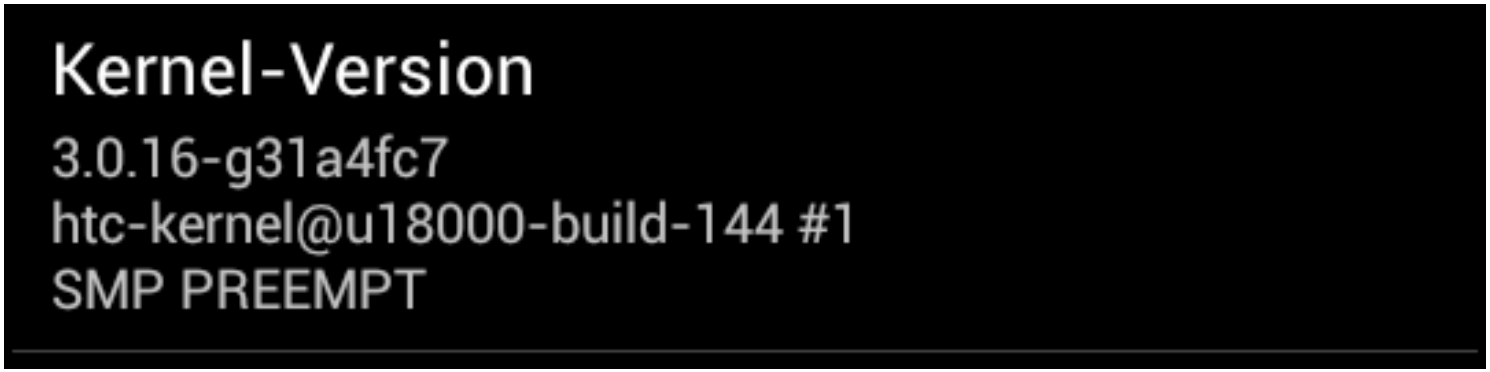
Laufzeitumgebung

- Entwicklungsumgebung
  - Ubuntu 12.04 LTS
- 3.2.0-23-realtime #36~ppa1-Ubuntu SMP **PREEMPT RT** Wed Apr 11 06:37:34 UTC 2012 i686 i686 i386 GNU/Linux
  - Interrupt Requests als Kernel-Threads
  - Harte Realtime Fähigkeit soll erreicht werden

Linux mit RT-Kernel



- Einsatz in mobilen Systemen.
  - z.B. in Smartphones



Linux mit RT-Kernel

- Cube Stormer II

Weltrekord im Lösen des Zauberwürfels

Linux mit RT-Kernel

- Seit Kernel 2.6.18
  - Einfließen des RT-Projektes in den offiziellen Kernel
  - Daher: Standardkernel hat bereits “gute” Echtzeiteigenschaften, wenn auch nicht unbedingt “Harte Echtzeitfähigkeit”
  - Ggf. reicht der Standardkernel absolut aus

Ausblicke

- Besseres Errorhandling
  - Überschreitung einer Periode
  - Fehlbenutzung
- Verbesserte Initalisierung der Threads
- Code-Generierung aus Prozessmodellen
- Autom. Analyse aller Prozesse bzgl. Ausführungszeit / Deadlines

Demo

< DEMO >

Fragen

- Fragen.....?