POSIX Echtzeit: Kernel 2.6 und Preempt-RT

Pengutronix.

Echtzeit-Systemplanung

- Wenn das zeitliche Verhalten spezifiziert ist, kann auch spezifiziert werden, welche Applikationsteile Echtzeit verlangen und welche nicht
  - Nicht alle anderen Teile der Applikation müssen ebenfalls Echtzeitverhalten besitzen, wenn es nur ein einzelner Teil erfordert!
  - Aufteilung der Aufgaben in Bereiche, die Echtzeit erfordern und die ohne Echtzeit auskommen
- Verteilung der Aufgabe in Bereiche im Kernel und Userspace
- Die Aufteilung der Aufgaben im Userspace muss nicht zwangsläufig zu getrennten Prozessen führen, sondern kann mittels Threads auch in einem Prozess realisiert werden
- Systeminterne Aufgaben (z.B. Interrupts) müssen auf Echtzeittauglichkeit und richtige Prioritäten geprüft sein

Pengutronix.

POSIX: Scheduler Prioritäten vergeben

Ändern der eigenen Priorität aus einem Programm heraus:

```
struct sched_param param;

[...]

param.sched_priority = -20;
pthread_setschedparam(thread, SCHED_OTHER, &param);
```

- Programme, die Prioritäten erhöhen, müssen als Root ausgeführt werden!
- Damit lassen sich Prioritäten unter der „Fair Queueing“ Strategie festlegen
- Dieses Beispiel erhöht nur die Priorität, verwendet aber weiterhin das „Fair Queueing“ Verfahren!
- Die möglichen Prioritäten liegen zwischen -20 (hoch) bis +19 (niedrig) („Nice-Level“)

POSIX: Scheduling Strategie ändern

Auf die gleiche Weise läßt sich die Scheduling Strategie ändern:

```
struct sched_param param;

[...]

param.sched_priority = 50;
pthread_setschedparam(thread, SCHED_RR, &param);
```

- Programme, die die Scheduler Strategie wechseln, müssen als Root ausgeführt werden!
- Dieses Beispiel erhöht die Priorität und wechselt in das „Priority Queueing“ Verfahren (Round Robin)
- Echtzeit-Prioritäten liegen zwischen 1 (niedrig) und 99 (hoch)

Wie steuert man den zeitlichen Ablauf?

- POSIX bietet mehrere Möglichkeiten für zeitliche Steuerung:
  - nanosleep(), clock_nanosleep()
  - Interval Timer
- Datenstruktur zur Definition von Zeiten:

```
struct timespec {
        time_t tv_sec;          /* seconds */
        long   tv_nsec;         /* nanoseconds */
};
```

- Beobachtung: Die zeitliche Auflösung in POSIX beträgt 1 ns
- Das bedeutet aber nicht, daß man Verzögerungen mit dieser Genauigkeit auch realisieren kann!

Verzögerung mit clock_nanosleep()

- Hochauflösende Sleeps mit spezifizierter Clock:

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
  const struct timespec *rqtp, struct timespec *rmtp);
```

- Mit `flags=TIMER_ABSTIME` kann festgelegt werden, ob `rqtp` ein Intervall (0) oder die absolute Zeit einer Clock (1) spezifiziert
- Funktion kann durch Signal unterbrochen werden
- In `rmtp` steht nach Ausführung der Funktion der Wert bis zum wirklichen Erreichen der Ablaufzeit, falls nicht NULL übergeben wurde
- Return Value: 0: Zeit ist korrekt abgelaufen

Pengutronix.

Clocks

- Es gibt mindestens folgende Clocks:
  - CLOCK_REALTIME: systemweite Echtzeituhr, nicht monoton
  - CLOCK_MONOTONIC: ist garantiert monoton
  - CLOCK_PROCESS_CPUTIME_ID: per Prozeß Timer der CPU
  - CLOCK_THREAD_CPUTIME_ID: Threadspezifische CPU Time
- Ermitteln der Auflösung einer Clock:

```
int clock_getres(clockid_t clk_id, struct timespec *res);
```

- Vorsicht: Auflösung sagt nichts über Echtzeiteigenschaften!

Verzögerungen mit clock_nanosleep()

- Verzögern des Programmablaufs um eine bestimmte Zeit:

```
[...]
/* change priority */
[....]

while (!endme) {

      /* do something */
      ....
      /* calc next wakeup */
      ....

      if (clock_nanosleep(id,TIMER_ABSTIME,&next,NULL))
            /* Fehlerbehandlung */
}
```

Periodischer Thread mit Interval Timer

- Periodisches Starten einer Programmfunktion:

```
void my_period_thread(union sigval val){
 [...] /* do something */
}

int main(){
     [...]
     struct itimerspec itimer = {
          .it_interval.tv_usec=500,
          .it_value.tv_sec=1
     };
     [...]
     timer_settime(timid,0,&itimer,NULL);
}
```

- Startet my_period_thread() periodisch unabhängig von dessen eigener Laufzeit (als eigenständigen Thread)

Probleme mit Echtzeit-Verhalten

- Obwohl ein RT Preempt System hinreichende Eigenschaften besitzt, um harte Echtzeit-Programme unter Linux zu implementieren, muß man aufpassen, daß der Determinismus nicht „beschädigt“ wird!
- Problem SMI: Moderne x86 Prozessoren haben System Management Mode Interrupt, der hochpriorer als der NMI ist!
- Problem Paging: Das Auslagern von Speicherseiten ins Swap oder Verwerfen von ReadOnly-Pages bedeutet, daß bei Bedarf die Seiten langsam zurückgelesen und die TLBs aktualisiert werden müssen.

  Abhilfe: mlock(), mlockall()

Determinismus erhalten

Unterteilen der gesamten Applikation in einen Teil mit Echtzeitanforderung und einen ohne.

Der Nicht-Echtzeitteil sollte:

- niemals Interrupte sperren
- möglichst keine Systemresourcen mit dem Echtzeitteil teilen, die jeweils nur exklusiv besessen werden dürfen

Der Echtzeitteil sollte:

- auf den kleinsten möglichen Programmteil eingeschränkt sein
- frei von Beschränkungen sein
- niemals andere Systemresourcen als Rechenleistung und Speicher verwenden
- keinen Speicher zur Laufzeit anfordern
- sich gegen Auslagern schützen

Entkoppeln

Trotzdem muß meistens zwischen beiden Programmteilen eine Kommunikation stattfinden.

Lösungsansatz: Asynchron entkoppeln

- mit den Mitteln der Inter Process Communication (IPC)
  - Messages
- POSIX Message Queues
  - Native Messages
- Ringpuffer
  - Shared Memory
    - Nachteil: Änderungen müssen per Polling detektiert werden

Messungen

- Testlauf auf einem Standard 2.6.18er Kernel (ixp425, big endian ARM)

```
$ cyclictest -p 80 -t 5 -n

T: 0 (  297) P:80 I:    1000 C:   34803 Min:    6376 Act:313224359
   Avg:156615364 Max:313224359
T: 1 (  298) P:79 I:    1500 C:   34803 Min:    6256 Act:295823221
   Avg:147914728 Max:295823221
T: 2 (  299) P:78 I:    2000 C:   34803 Min:    6236 Act:278422198
   Avg:139214206 Max:278422198
T: 3 (  300) P:77 I:    2500 C:   34803 Min:    6223 Act:261021182
   Avg:130513690 Max:261021182
T: 4 (  301) P:76 I:    3000 C:   34803 Min:    6209 Act:243620166
   Avg:121813175 Max:243620166
```

- I: Soll-Verzögerung in µs
- C: Anzahl Durchläufe
- Min: Minimale Abweichung (von der Sollzeit)
- Act: Aktuelle Abweichung
- Avg: Durchschnittliche Abweichung
- Max: Maximale Abweichung

Messungen

- Testlauf auf einem RT-Preempt 2.6.18er Kernel mit HR-Timer (ixp425, big endian ARM)

```
$ cyclictest -p 80 -t 5 -n -l 10000

T: 0 (  253) P:80 I: 1000 C: 10000 Min:  18 Act:  22 Avg:  25 Max:  138
T: 1 (  254) P:79 I: 1500 C:  6668 Min:  21 Act:  27 Avg:  25 Max:  129
T: 2 (  255) P:78 I: 2000 C:  5001 Min:  22 Act:  57 Avg:  26 Max:  133
T: 3 (  256) P:77 I: 2500 C:  4001 Min:  20 Act:  46 Avg:  25 Max:  116
T: 4 (  257) P:76 I: 3000 C:  3334 Min:  20 Act:  88 Avg:  24 Max:   97
```

- I: Soll-Verzögerung in µs
- C: Anzahl Durchläufe
- Min: Minimale Abweichung (von der Sollzeit)
- Act: Aktuelle Abweichung
- Avg: Durchschnittliche Abweichung
- Max: Maximale Abweichung

Zusammenfassung

- Soft-Realtime wird mit ausreichender Auflösung von der POSIX API bereitgestellt
- Es sind keine besonderen Umgebungen/Werkzeuge/Bibliotheken erforderlich, um davon Gebrauch zu machen
- POSIX konforme Programme können ohne Änderungen weiter verwendet werden
- Preempt-RT implementiert auch die Genauigkeit für HRT und liefert die PI Mutexe
- Echtzeit-Eigenschaften können nachträglich hinzugefügt werden, teilweise ohne das Programm anpassen zu müssen
- Echtzeit-Eigenschaften lassen sich aus dem Userspace heraus verwenden, aber man muß aufpassen, was man tut