Parallele Programmierung in eingebetteten Systemen

David Hobach, Ruhr-Universität Bochum (28.07.2009)

Zusammenfassung—**Der aktuelle Trend der CPU-Industrie ist der Einsatz von immer mehr Kernen bei gleichbleibender oder gar sinkender Taktfrequenz. Da auch im Bereich von eingebetteten Systemen immer leistungsstärkere und energieeffizientere CPUs benötigt werden (u.a. bei Echtzeitanforderungen), macht dieser Trend auch dort nicht halt. Das große Problem ist dabei die effiziente Nutzung der zur Verfügung gestellten Kerne durch den Programmierer. Einige Möglichkeiten sollen hier vorgestellt werden.**

I. Einleitung

DER größte Vorteil von multicore-Architekturen in eingebetteten Systemen liegt in der Energieeffizienz: Untertaktet man einen Kern um 20%, so ergibt sich ein Leistungsverlust von 13% bei einer Energieeinsparung von 50%. Daraus folgt, dass der Einsatz von zwei Kernen bei jeweils 80% des Taktes des einzelnen Kernes 73% mehr Leistung bei derselben Energieaufnahme bedeutet [RO08]. Da laut [TH09] im Jahre 2007 rund 5 Milliarden Prozessoren weltweit genutzt wurden (davon weniger als 1% "sichtbar" in PCs/Workstations), ergibt sich ein gigantisches Potential. Ein Nachteil des Einsatzes mehrerer Kerne ist der zusätzliche Platzbedarf auf dem Chip, der dafür meist aber auch bessere Eigenschaften bezüglich der Wärmeleitfähigkeit besitzt.
Das größte Problem jedoch liegt in dem Ausnutzen der Hardware durch die Software. Code, der ursprünglich für Einzel-Kerne geschrieben wurde, soll nun mehrere Kerne ausnutzen: Dies lässt sich entweder automatisch durch gewisse Compiler-Maßnahmen oder explizit auf das Programm zugeschnitten und damit meist auch effizienter durch den Programmierer bewerkstelligen.
In dieser Abhandlung werden nach einem Überblick über die Parallele Programmierung im Allgemeinen genau diese beiden Alternativen betrachtet. Dabei ist das explizite Ausprogrammieren von Threads auch im Bereich von Systemen mit Einzel-Kernen von Relevanz, da häufig nebenläufig gearbeitet werden muss, d.h. dass eine Aufgabe (z.B. Videowiedergabe bei einem Smartphone) eine andere nicht "blockieren" darf (z.B. Darstellung des GUI). Zu guter Letzt wird noch der Einsatz paralleler Programmierung unter Echtzeitbedingungen betrachtet.

II. Parallele Programmierung im Allgemeinen

Da eine CPU pro Taktzyklus maximal eine Instruktion abarbeiten kann, existiert "echte" Parallelität nur beim Einsatz von multicore-Architekturen. Der Einsatz von Prozessen ermöglicht jedoch eine Art "virtuelle" Parallelität: Die CPU widmet sich zunächst einigen Instruktionen des einen Prozesses, dann werden sämtliche Register, die Prozessor-Flags sowie der Instruction Pointer (der Kontext des ersten Prozesses) im RAM gesichert und der Kontext des zweiten Prozesses geladen, um einige seiner Instruktionen auszuführen ("context switching"). Die CPU wird sozusagen geleert, bevor der nächste Prozess ausgeführt wird [LB03]. Der sogenannte "Scheduler" bestimmt dabei die Reihenfolge, in welcher die Aufgaben abgearbeitet werden. Er ist Teil des eingesetzten Betriebssystems und kann für gewöhnlich stets von Interrupts, also plötzlich auftretenden Signalen, die eine sofort abzuarbeitende Aufgabe signalisieren, unterbrochen werden. Ein einfaches Beispiel für einen Scheduler ist der häufig benutzte Round-Robin Scheduler, der in einer FIFO-Warteschlange jeder Aufgabe ein Zeitintervall Δt garantiert. Reicht dieses Intervall nicht aus, um die Aufgabe auszuführen, wird ihr Kontext wieder gespeichert und sie wird ans Ende der Warteschlange gesetzt [IRM02]. Da einige Aufgaben wichtiger sein können als andere, werden ihnen Prioritäten zugewiesen, die der Scheduler je nach Implementierung beachten muss. Vor allem unter Echtzeitbedingungen spielt dies eine Rolle.
Bei Threads - der hier am häufigsten betrachteten Struktur - handelt es sich um eine Art "leichtgewichtige" Prozesse. Sie werden einem Prozess untergeordnet, weswegen ihre Verwaltung wesentlich weniger komplex ist als die eines ganzen Prozesses (Prozess-ID, Heap, shared libraries etc. müssen nicht verwaltet werden). Zudem teilen sich Threads mit dem Prozess, dem sie zugeordnet sind, den zugewiesenen Arbeitsspeicher, was die Kommunikation stark vereinfacht. Sie können jedoch vom Scheduler wie Prozesse behandelt werden. Parallelisierung innerhalb eines Programmes erfolgt folglich meist durch das Aufteilen einer Aufgabe auf verschiedene Threads, die Parallelisierung mehrerer Programme durch die Bildung von Prozessen. Das Aufteilen einer Aufgabe kann entweder durch die Aufteilung der zugehörigen Daten in Partitionen, deren Abarbeitung voraussichtlich gleich viel CPU-Zeit benötigen wird, oder durch Aufteilung in gewisse Funktionsblöcke erfolgen. In jedem Fall muss das zu lösende Problem eine der beiden Aufteilungen unterstützen, um parallelisierbar zu sein [BARN09]. Je nach Problemstellung ist es auch nötig, dass die Threads miteinander kommunizieren oder synchronisiert werden. Soll die Arbeit der einzelnen Threads beispielsweise am Ende zusammengeführt werden, so muss eine Art "Barriere" eingeführt werden, ab der jeder Thread auf alle anderen wartet. Bei der Synchronisation zwischen zwei oder mehreren Threads werden zudem sogenannte Semaphoren eingesetzt, welche verhindern sollen, dass mehr als eine bestimmte Anzahl von Threads gleichzeitig auf ei-

ne gemeinsam genutzte Resource zugreifen. Darf nur ein Thread zu einem Zeitpunkt auf eine gemeinsam genutzte Resource zugreifen, so spricht man von einem "Mutex" ("mutual exclusion") [BARN09]. Semaphoren und Mutexe werden meist in Form simpler Zähler implementiert, die den Zugriff auf die Resource nur erlauben, wenn der Zähler unter dem gewünschten Maximalwert liegt.

III. Automatische Parallelisierung

Automatische Parallelisierung meint den Einsatz von Tools, meist speziellen Compilern, zum Generieren von parallelem Code aus seriellem. Der Vorteil ist offensichtlich: Es ist kein Arbeitseinsatz nötig. Ziel der Parallelisierung sind zum Einen Schleifen, deren Daten sich leicht auf mehrere Prozesse aufteilen lassen. Allerdings kann der Compiler auch zu fehlerhaften Schlüssen kommen, was im schlimmsten Fall sogar zu einer Verschlechterung der Performanz führen kann [BARN09]. Viele moderne Compiler versuchen auch, den generierten Code so umzustrukturieren, dass aufeinanderfolgende Instruktionen von Prozessoren mit mehreren ALUs ("superscalar CPUs") gleichzeitig ausgeführt werden können [EI05]. Dies kann man z.B. bei folgendem i386-Code aus der Windows-Kernel-Dll "ntdll" betrachten:

```
1  push ebx
2  push esi
3  mov esi, dword ptr [ecx+10]
4  push edi
5  mov edi, dword ptr [ebp+C]
6  cmp edi, −1
7  lea ebx, dword ptr [edi+1]
8  je short ntdll.7C962559
9  cmp ebx, edx
10 ja short ntdll.7C962559
```

Der bedingte Sprung in Zeile 8 bezieht sich auf den Vergleichsbefehl aus Zeile 6. Zeile 7 wurde jedoch eingefügt, um den Befehl aus Zeile 6 gleichzeitig zu dem aus Zeile 7 ausführen zu können und nicht auf das Ergebnis der Vergleichsoperation warten zu müssen. Es handelt sich um sogenannten "interleaved code" [EI05].
Selbstverständlich ist es auch möglich, auf einem wesentlich höheren Abstraktionsniveau automatisch zu parallelisieren. So kann man dem Compiler beispielsweise über Präprozessor-Anweisungen klar machen, was wie parallelisiert werden soll (vgl. OpenMP, semi-manuell) [BARN09]. Bislang gilt dabei: Will man applikationsspezifisch optimieren, so muss man manuell parallelisieren. Daher wird im Folgenden vor allem die manuelle Parallelisierung behandelt.

IV. Manuelle Parallelisierung

Im Gegensatz zur automatischen Parallelisierung liegt bei der manuellen Parallelisierung der Großteil der Verantwortung beim Programmierer. Er muss erkennen können, welche Teile des zu lösenden Problems sich parallelisieren lassen und diese Parallelisierung ausprogrammieren. Dies geschah ursprünglich mithilfe von hardware-spezifischen APIs, die von den Verkäufern der entsprechenden CPUs implementiert wurden. Inzwischen existieren jedoch einige Standards, welche von vielen Hardwareherstellern implementiert wurden: Darunter finden sich die POSIX Threads und OpenMP zum Erstellen und Synchronisieren von Threads sowie MPI zur Kommunikation zwischen Threads [BARN09]. Diese setzen natürlich ein Betriebssystem zur Unterstützung von Prozessen, Threads, Scheduling etc. auf dem eingebetteten System voraus.

A. POSIX Threads

Die POSIX Threads, kurz auch Pthreads genannt, wurden vom IEEE im Jahre 1995 mit dem Standard 1003.1c spezifiziert. Der Standard war für die Programmiersprache C gedacht und sollte in Form einer Bibliothek implementiert werden. Inzwischen sind Pthreads z.B. Bestandteil der libc [BARN09].
Das folgende Beispiel aus [BARN09] soll einen kurzen Einstieg in die Programmierung mit Pthreads gewähren:

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define NUM_THREADS     2
5
6  char *messages[NUM_THREADS];
7
8  void *PrintHello(void *threadid)
9  {
10 int *id_ptr, taskid;
11 id_ptr = (int *) threadid;
12 taskid = *id_ptr;
13 printf("Thread %d: %s\n", taskid,
       messages[taskid]);
14 pthread_exit(NULL);
15 }
16
17 int main(int argc, char *argv[])
18 {
19 pthread_t threads[NUM_THREADS];
20 int *taskids[NUM_THREADS];
21 int rc, t;
22
23 messages[0] = "English: Hello World!";
24 messages[1] = "French: Bonjour, le monde!";
25
26 for(t = 0; t < NUM_THREADS; ++t)
27         {
28         taskids[t] = (int *)
               malloc(sizeof(int));
29         *taskids[t] = t;
30         printf("Creating thread %d\n", t);
31         rc = pthread_create(&threads[t],
              NULL, PrintHello, (void *)
               taskids[t]);
32         if (rc)
33         {
34         printf("ERROR; return code from
               pthread_create() is %d\n", rc);
35         exit(-1);
36         }
37         }
38 pthread_exit(NULL);
39 }
```

Dem geneigten Leser fällt zunächst der Funktionsaufruf in Zeile 31 auf, der einen Thread erstellt: Übergeben wird ein Pointer auf eine Struktur vom Typ pthread_t zur eindeutigen Identifizierung des Threads, ein Pointer auf die zu verwendenden Thread-Attribute (pthread_attr_t) oder NULL (hier), falls die default-Werte verwendet werden sollen, ein Pointer auf eine Routine mit Rückgabewert void*,

welche der Thread nach seiner Erstellung aufrufen soll, sowie Pointer auf ein Argument für ebendiese Routine, in unserem Fall willkürlich gewählte Task IDs. Sollen mehrere Argumente übergeben werden, so müssen diese in einer struct gebündelt werden. Die pthread_exit()-Routine in den Zeilen 14 und 38 beendet den laufenden Thread und ersetzt das return. Ein Aufruf von pthread_exit() anstatt von return ist auch in main() sinnvoll, da ansonsten alle Threads, die innerhalb von main() erstellt wurden, nach dessen Beendigung zerstört würden.
Das Beispiel verdeutlicht auch, dass jedem Thread konsistente Argumente übergeben werden sollten, d.h. es wäre falsch, Zeile 31 durch

```
rc = pthread_create(&threads[t], NULL,
     PrintHello, (void *) &t);
```

zu ersetzen. Dann wäre die Variable t eventl. schon wieder geändert worden, wenn der Thread nach seiner Erstellung versucht, darauf zuzugreifen.
Die Pthreads API ermöglicht zudem die Synchronisation zwischen Threads über Mutexe und sogenannte "Condition Variables", welche ermöglichen, dass ein Thread auf eine von einem anderen Thread herbeigeführte Bedingung warten kann. Auch das allgemeine Warten auf die Beendigung eines Threads ("join") wird unterstützt. Die Kommunikation zwischen den Threads kann schlichtweg über den gemeinsam genutzten Hauptspeicher erfolgen [BARN09].
Insgesamt bieten die POSIX Threads also viele Methoden zum expliziten und feinen Ausprogrammieren von Parallelität.

B. OpenMP

OpenMP (Open Multi-Processing) ist eine von verschiedenen Hardware- und Softwareherstellern (u.a. Intel, Sun, IBM, HP) im Jahre 1997 definierte API. Sie wurde für die Programmiersprachen C/C++ und Fortran spezifiziert. Es wird dabei wie bei den Pthreads von einer Architektur mit mehreren Prozessorkernen, aber gemeinsam genutztem Hauptspeicher (SMP-Architektur) ausgegangen. Bei OpenMP handelt es sich um eine semi-manuelle Parallelisierung, denn der Programmierer gibt dem Compiler über Direktiven Anweisungen, was und wie parallelisiert werden soll. Um in dem Programm herauszufinden, wie viele Threads gestartet wurden oder um Threads zu synchronisieren o.ä., kann die OpenMP-Bibliothek verwendet werden. Eine weitere Möglichkeit, um z.B. die maximale Anzahl zu erstellender Threads festzulegen, besteht im Setzen systemweiter Umgebungsvariablen [BARN09].
OpenMP wird stetig weiterentwickelt; im Jahre 2008 wurde die Version 3.0 spezifiziert.
Das hinter OpenMP stehende Modell ist das sogenannte Fork-Join-Modell: Zunächst führt ein "master thread" (main()) sequentiell Aufgaben durch, bis eine Compiler-Direktive erreicht wird, welche eine parallel abzuarbeitende Region ankündigt. Daraufhin erstellt der master thread Kopien von sich selbst ("fork"), mit denen er zusammen die parallele Region abarbeitet. Am Ende dieser Region wird gewartet, bis alle Threads ihren Teil abgearbeitet haben ("join"), die Threads synchronisieren sich ggf. und terminieren, so dass nur der master thread übrig bleibt [BARN09].
Der folgende Code soll die Funktionsweise von OpenMP näher verdeutlichen:

```
1  #include <omp.h>
2  #define CHUNKSIZE 100
3  #define N       1000
4
5  int main(int argc, char *argv[])
6  {
7  int i, tid, chunk;
8  float a[N], b[N], c[N];
9
10 /* some initializations */
11 for (i=0; i < N; ++i) a[i] = b[i] = i * 1.0;
12 chunk = CHUNKSIZE;
13
14 #pragma omp parallel shared(a,b,c,chunk)
       private(i,tid)
15         {
16         tid = omp_get_thread_num();
17         if (tid != 0) printf("Hello from
               thread %d.\n",tid);
18         else printf("Hello from the master
               thread.\n");
19
20         #pragma omp for
               schedule(dynamic,chunk) nowait
21         for (i=0; i < N; ++i) c[i] = a[i] +
               b[i];
22         }
23 return 0;
24 }
```

Bis Zeile 13 führt der master thread sämtliche Instruktionen alleine und sequentiell aus. Von Zeile 14 bis Zeile 22 (Klammern beachten!) wird parallel gearbeitet: Alle Threads teilen sich dabei die Variablen a, b und c sowie chunk (shared), von den Variablen i und tid dagegen besitzt jeder Thread eine eigene Kopie (private). omp_get_thread_num() aus Zeile 16 ist eine in omp.h deklarierte Routine, die die jeweilige Thread ID zurückgibt. Dem master thread wird dabei stets die Null zugewiesen. In Zeile 20 wird schließlich spezifiziert, wie die mit der folgenden For-Schleife zu erledigende Arbeit unter den Threads aufgeteilt werden soll (schedule): Jedem Thread soll ein Teil der Schleife der Größe chunk zugewiesen werden. Nachdem ein Thread seine Arbeit abgeschlossen hat, wird ihm ein weiterer Teil zugewiesen (dynamic). Sollte keine Arbeit mehr vorhanden sein, so soll ein Thread nicht auf andere Threads warten/sich mit diesen synchronisieren, sondern schlichtweg terminieren (nowait).
Dieses schlichte Beispiel macht schnell klar, dass mit OpenMP bestehender, sequentiell arbeitender Code recht einfach durch Hinzufügen von Compiler-Direktiven parallelisiert werden kann. Falls der Einsatz von Bibliotheksroutinen nicht notwendig sein sollte, so kann der geschriebene parallele Code sogar weiterhin mit Compilern übersetzt werden, die OpenMP nicht unterstützen, da die Compiler-Direktiven schlichtweg ignoriert werden. Dadurch dass der Programmierer die konkrete Parallelsierung dem Compiler überlässt, verliert er im Vergleich zu Pthreads die Kontrolle über einige Details.

C. MPI

Während Pthreads und OpenMP von einer Architektur mit gemeinsamem Hauptspeicher ausgehen, geht man bei MPI (Message Passing Interface) von einer Architektur mit mehreren CPUs aus, die jeweils über eigenen Hauptspeicher verfügen, der sich nicht im Adressraum der anderen CPUs befindet ("distributed memory"). Diese CPUs sind über einen nicht näher spezifizierten Datenbus miteinander verbunden und sollen zusammen eine Aufgabe bewältigen. Um dies zu ermöglichen, müssen die Threads miteinander kommunizieren können - hier kommt MPI ins Spiel: MPI ist eine in den Jahren 1994 (MPI-1) und 1996 (MPI-2) von über 40 verschiedenenen teilnehmenden Organisationen festgelegte Spezifikation zum Austausch von Nachrichten zwischen Threads. Wie auch OpenMP ist MPI kein offizieller IEEE oder ISO-Standard, sondern vielmehr ein Industriestandard. Inzwischen existieren eine Vielzahl von Implementierungen in Form von Bibliotheken für diverse Programmiersprachen [BARN09].
Die Spezifikation selbst beschreibt keine Methoden, um Threads zu erstellen. Dies geschieht durch ein Programm (z.B. "mpirun"), welches Kopien des auszuführenden Programms auf die zur Verfügung stehenden Prozessoren verteilt. Aus Effizienzgründen wird normalerweise pro Prozessor nur eine Kopie vergeben (MPI wird häufig in hochperformanten Rechenclustern eingesetzt).
Im Folgenden wird ein kurzes Beispiel betrachtet [BARN09]:

```
1  #include "mpi.h"
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6  int numtasks, rank, next, prev, buf[2],
       tag1=1, tag2=2;
7  MPI_Request reqs[4];
8  MPI_Status stats[4];
9
10 MPI_Init(&argc,&argv);
11 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
12 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13
14 prev = rank-1;
15 next = rank+1;
16 if (rank == 0)   prev = numtasks - 1;
17 if (rank == (numtasks - 1))   next = 0;
18
19 MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1,
       MPI_COMM_WORLD, &reqs[0]);
20 MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2,
       MPI_COMM_WORLD, &reqs[1]);
21 MPI_Isend(&rank, 1, MPI_INT, prev, tag2,
       MPI_COMM_WORLD, &reqs[2]);
22 MPI_Isend(&rank, 1, MPI_INT, next, tag1,
       MPI_COMM_WORLD, &reqs[3]);
23
24 /* do some work here */
25
26 MPI_Waitall(4, reqs, stats);
27 MPI_Finalize();
28 return 0;
29 }
```

Zunächst wird die MPI-Umgebung mit den dem Programm übergebenen Parametern initialisiert (Zeile 10). Danach wird in Zeile 11 die Anzahl der erstellten Threads/Prozesse bestimmt. Bei MPI_COMM_WORLD handelt es sich dabei um die Gruppe von Prozessen, die alle umfasst. Der Programmierer kann mit MPI allerdings auch eigene Untergruppen bilden und beispielsweise nur innerhalb von diesen eine Kommunikation durchführen. Der Rang (Zeile 12) eines Prozesses ist seine einzigartige Task-ID pro Gruppe. Er kann wie in den Zeilen 19 bis 22 realisiert zur Kommunikation mit anderen Prozessen verwendet werden. Bei MPI_Isend bzw. MPI_Irecv handelt es sich um nichtblockierende Sende- bzw. Empfangsoperationen, d.h. der jeweils nächste Befehl wird ggf. schon ausgeführt bevor das mit der aufgerufenen Funktion gesetzte Ziel erreicht wurde. Der erste Parameter gibt bei all diesen Funktionen einen Puffer an, in dem zu sendende Daten zu finden sind bzw. zu empfangende Daten gespeichert werden können. Der zweite Parameter gibt die Größe des Puffers an, der dritte den Datentyp. Der Datentyp wird benötigt, weil MPI eigene Datentypen definiert. Der vierte Parameter gibt den Rang des Ziels bzw. der Quelle der Operation an, der fünfte eine der Übertragung der Nachricht zugeordnete, möglichst eindeutige Zahl zur Identifikation. Darauf folgt noch die Gruppe, in welcher die Nachricht verschickt/empfangen werden soll, sowie ein Handle, welches benutzt werden kann, um das Abschließen der entsprechenden Operation abzuwarten. Genau dies geschieht in Zeile 26: Es wird darauf gewartet, dass alle 4 Operationen abgeschlossen werden. Fehler o.ä. werden in der jeweiligen Status-Variable dokumentiert. Zu guter Letzt (Zeile 27) wird noch sämtlicher mit MPI verbundener Speicher freigegeben. Insgesamt gesehen implementiert der vorliegende Code eine Ringtopologie.
Neben diesen Methoden bietet MPI noch eine Vielzahl weiterer, um Prozesse oder Threads virtuell in Topologien anzuordnen oder zu synchronisieren.

V. Parallele Programmierung unter Echtzeitbedingungen

Im Bereich eingebetteter Systeme sind häufig Echtzeitbedingungen einzuhalten: Eine Aufgabe muss bis zu einem vorgegebenen Zeitpunkt erledigt sein, sonst ist ihr Ergebnis wertlos (harte Echtzeitbedingungen, z.B. im Auto oder Flugzeug) oder zumindest weniger nützlich (weiche Echtzeitbedingungen, z.B. beim Videostreaming). Um Echtzeitbedingungen einhalten zu können, wird die Unterstützung des Betriebssystems benötigt: Es muss einen Scheduler besitzen, der mit priorisierbaren Tasks umgehen kann, exakte Zeitmessungen durchführen können und jegliche vom Betriebssystem zur Verfügung gestellten Funktionalitäten müssen deterministisch in der Zeit ausführbar sein. Erfüllt ein Betriebssystem diese Bedingungen, so spricht man von einem "Real-Time Operating System" (RTOS) [OB01]. Einige sind z.B. freeRTOS, ThreadX, QNX, LynxOS oder Nucleus RTOS. Das RTOS muss beim Scheduling ggf. auch die unter ihm liegende Architektur (ein oder mehrere Kerne) berücksichtigen, da es in multicore-Architekturen beispielsweise auch vorteilhaft sein kann, zwei Prozesse auf derselben CPU laufen zu lassen, falls diese auf dieselben Speicherbereiche zugreifen sollten und somit den L2/3-Cache optimal gemeinsam nutzen könnten.

Viele RTOS-Scheduler gehen so vor: Der Prozess mit der höchsten Priorität bekommt die CPU zugeordnet; sollten mehrere Prozesse mit gleicher Priorität in der Warteschlange vorhanden sein, so wird zwischen diesen ein Round-Robin Scheduling (vgl. Abschnitt II) durchgeführt [IRM02]. Nun stellt sich noch die Frage, welche Prioritäten man welcher Real-Time-Aufgabe zuweisen soll. Bei statisch zugewiesenen Prioritäten (Priorität eines Threads/Prozesses bleibt während seiner Lebenszeit konstant) und mit gegebener Periode wiederkehrenden Aufgaben (Deadline vor Anfang der nächsten Periode) ist es sinnvoll, den sogenannten "Rate Monotonic Algorithm" (RMA), einen einfachen Greedy-Algorithmus, anzuwenden: Der Aufgabe mit der kleinsten Periode wird die größte Priorität zugeordnet [SB02]. Dieser Algorithmus ist bei statisch gesetzten Prioritäten nachweisbar optimal. Sei n die Anzahl von Tasks und W_n die untere Grenze der anteilhaften CPU-Auslastung durch RMA angesetzte Tasks, so gilt: $W_n = n * (2^{1/n} - 1)$. RMA garantiert also, dass die Deadlines aller Tasks, deren Summe der CPU-Nutzung $U_i = \frac{C_i}{T_i}$ (C_i: worst case execution time von Task i, T_i: Periode von Task i) W_n nicht überschreitet, eingehalten werden können. In Spezialfällen ist es noch möglich, die CPU besser auszunutzen ohne Deadlines zu überschreiten [SB02].

Werden Prioritäten dynamisch vergeben, d.h. darf sich die Priorität nach bestimmten Algorithmen zur Laufzeit ändern, so kann eine weitaus bessere CPU-Ausnutzung erreicht werden, das System wird allerdings je nach Ziel auch deutlich komplexer. Will man z.B. verhindern, dass Prozesse mit hoher Priorität zu lange ausgeführt werden, so kann man deren Priorität nach einer gewissen Zeiteinheit verringern [SB02], [IRM02].

Ein häufig größeres Problem als das der CPU-Auslastung ist beim Scheduling in Echtzeitsystemen das der Prioritäts-Inversion: Teilen sich ein Task H (hohe Priorität) und ein Task N (niedrige Priorität) eine gemeinsame Resource, die vor der Erstellung von Task H von Task N über eine Semaphore o.ä. beschlagnahmt wurde, so muss die höher-priore Aufgabe auf die unwichtigere Aufgabe warten. Dies ist der Normalfall und auch noch kein Problem. Das eigentliche Problem entsteht erst dann, wenn ein Task M (mittlere Priorität) den Task N auf unbestimmte Zeit unterbricht (vgl. Abb. 1, rot: aktive Aufgabe). In diesem Fall spricht man von Prioritätsinversion, da eine wichtige Aufgabe von einer mittelmäßig wichtigen verdrängt wurde [KB02]. Eine Lösung für dieses Problem ist das Vererben der höheren Priorität von Task H an Task N während der Nutzung der Resource, was die Unterstützung des Betriebssystems voraussetzt. Alternativ kann die Resource mit einer Priorität versehen werden, die der höchsten Priorität des auf sie zugreifenden Tasks entspricht. Der Task, der die Resource inne hat, erhält dann die Priorität der Resource plus Eins [KB02]. Es gilt zu beachten, dass beide Lösungen die dynamische Vergabe von Prioritäten voraussetzen. In multicore-Systemen bestünde eine weitere Lösung darin, Task M auf einen anderen Kern zu übertragen ("thread migration").

Standards zur Parallelen Programmierung unter Echtzeitbedingungen finden sich beispielsweise in den Erweiterungen b, d und j zum POSIX-Standard 1003.1. Deren Implementierung wurde in [OB01] getestet und für ausreichend empfunden, wenngleich für die Praxis empfohlen wird, mindestens eine CPU für Real-Time-Aufgaben zu reservieren.

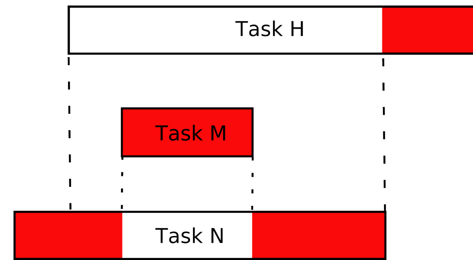

Abbildung 1
PRIORITÄTS-INVERSION

VI. ZUSAMMENFASSUNG

Die vorhandenen Methoden und Standards zur Beschreibung paralleler Systeme scheinen ausreichend für die meisten Anwendungsbereiche. Entwicklungsbedarf besteht dagegen noch im Bereich der Scheduler bzw. der damit verbundenen Betriebssysteme. Aktuell beachten viele Scheduler nur die Nutzung der CPU, des Speichers und von I/O-Operationen. Je nach Applikation ist beispielsweise aber ebenfalls eine Berücksichtigung des Energieverbrauchs wünschenswert. Derzeit geht der Trend auch in Richtung flexiblerer Scheduler, die sich zur Laufzeit ändernde Anforderungen an Tasks berücksichtigen (Änderungen z.B. durch Umwelteinflüsse). Weitere Ideen sind, jedem Task einen gewissen festen Anteil an der CPU-Zeit während seiner Laufzeit zuzusichern, mehrere Scheduler je nach Aufgabe gleichzeitig einzusetzen oder das Betriebssystem Komponenten-basierter zu machen, so dass der Scheduler leichter ausgetauscht werden kann und das OS schneller zu anderen Plattformen portiert werden kann [BUT07].

LITERATUR

[BARN09] Blaise Barney. Lawrence Livermore National Laboratory. Introduction to Parallel Computing, POSIX Thread Programming, OpenMP, MPI. 2009. URL: https://computing.llnl.gov/?set=training&page=index#training_materials

[IRM02] Prof. Dr. K. Irmscher. Prozessverwaltung. 2002. URL: http://www.munz-udo.de/pdf_files/betriebssysteme/prz_sc02.pdf

[LB03] Jean Labrosse and Michael Barr. Introduction to Preemptive Multitasking. 2003. URL: http://www.netrino.com/Embedded-Systems/How-To/RTOS-Preemption-Multitasking

[KB02] David Kalinsky and Michael Barr. Introduction to Priority Inversion. 2002. URL: http://www.netrino.com/Embedded-Systems/How-To/RTOS-Priority-Inversion

[SB02] David Stewart and Michael Barr. Introduction to Rate Monotonic Scheduling. 2002. URL: http://www.netrino.com/Embedded-Systems/How-To/RMA-Rate-Monotonic-Algorithm

[RO08] Philip Ross. Why CPU Frequency Stalled. 2008. URL: http://www.spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled

[TH09] Dr. Wolfgang Theimer. Embedded Multimedia. Vorlesung 2. Ruhr-Universität Bochum. 2009.

[EI05] Eldad Eilam. Reversing - Secrets of Reverse Engineering. S. 156. Wiley Publishing. 2005.

[OB01] Kevin M. Obenland. POSIX in Real-Time. 2001. URL: http://www.embedded.com/story/OEG20010312S0073

[BUT07] Giorgio Buttazzo. Research Trends in Real-Time Computing for Embedded Systems. 2007. URL: http://www.cs.aau.dk/~bt/DAT5E07/Henrik2.pdf