Parallele Programmierung in eingebetteten Systemen

Seminar Integrierte Schaltungen und Systeme für schnelle Datenübertragung im Internet

David Hobach
Betreuer: Shadi Traboulsi

Ruhr-Universität Bochum

25.07.2009 / Bochum

Outline

Motivation (1)

Trend: Multicore

Ziele bei eingebetteten Prozessoren:

- Leistung
- Energieeffizienz
- Wärmeleitfähigkeit
- Größe

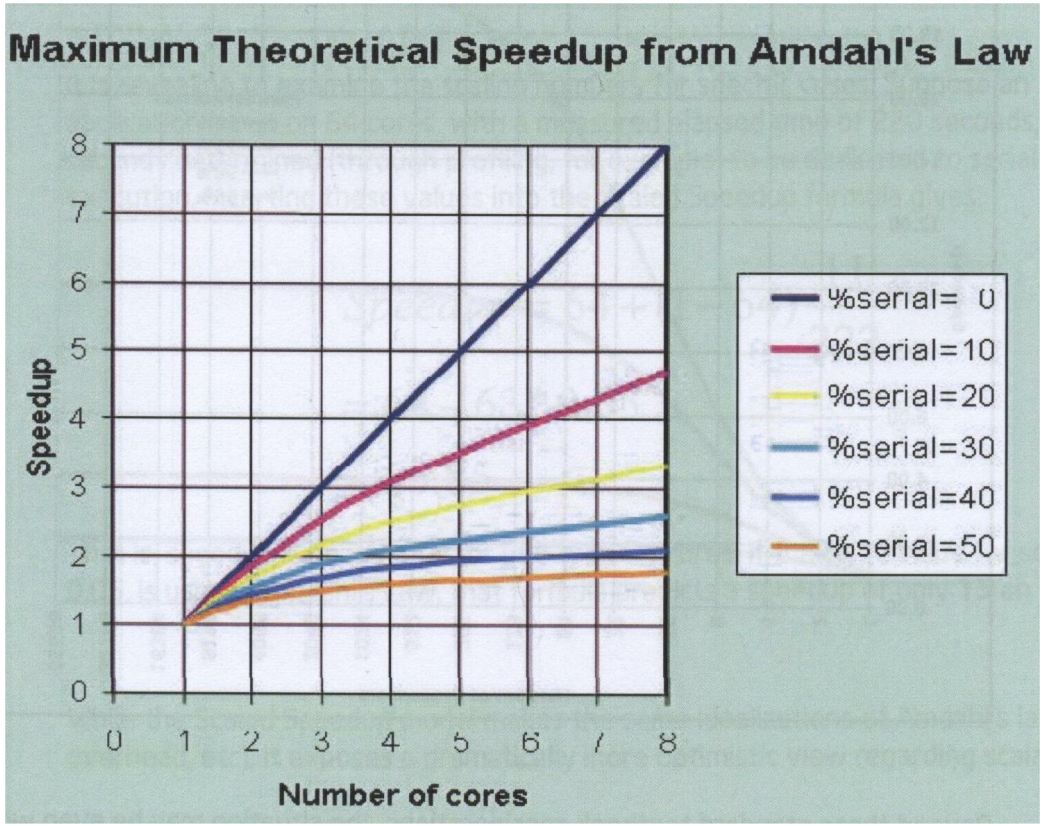
$\Rightarrow$ Multicore-Architekturen:

- 20% Untertaktung ergibt einen Leistungsverlust von 13% bei Energieeinsparungen von 50% [1]
  $\Rightarrow$ mehrere Kerne
- Nachteil: Platzbedarf
- Problem: effizientes Ausnutzen der Kerne $\Rightarrow$ Parallele Programmierung

[1] Philip Ross. IEEE. Why CPU Frequency Stalled. 2008.

Motivation (2)

Potential - Amdahls Gesetz

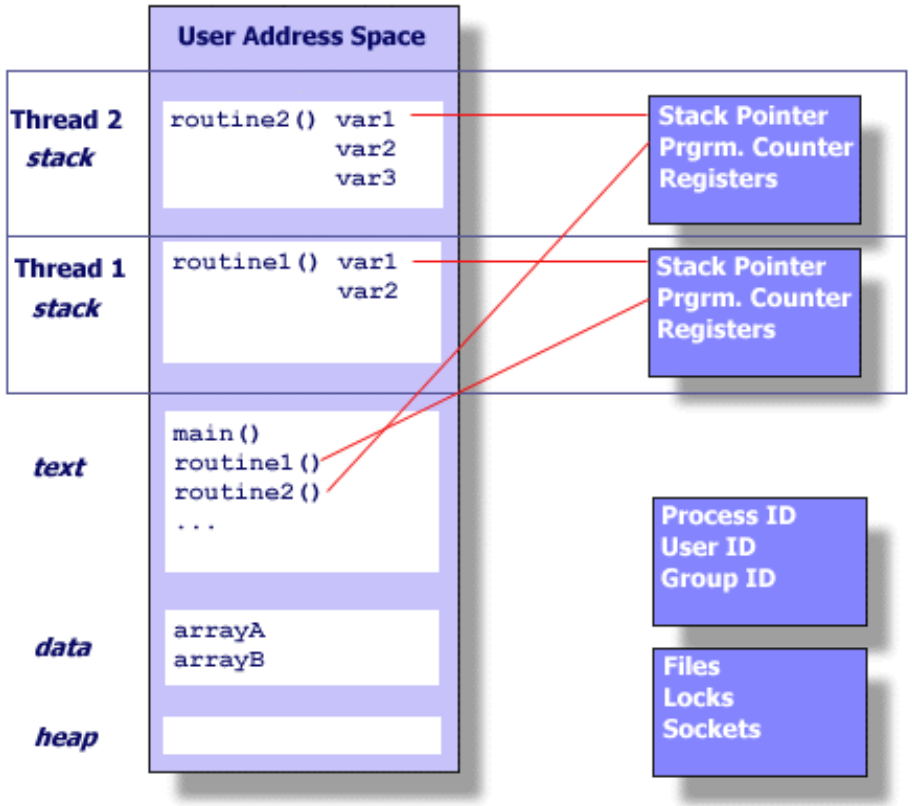


- $Speedup(p) = \frac{1}{s+(1-s)/p}$ (p: Anzahl Kerne, s: serieller Anteil)
- optimistisch: kein Overhead durch Thread-Erstellung, Synchronisation, Kommunikation...

Überblick (1)

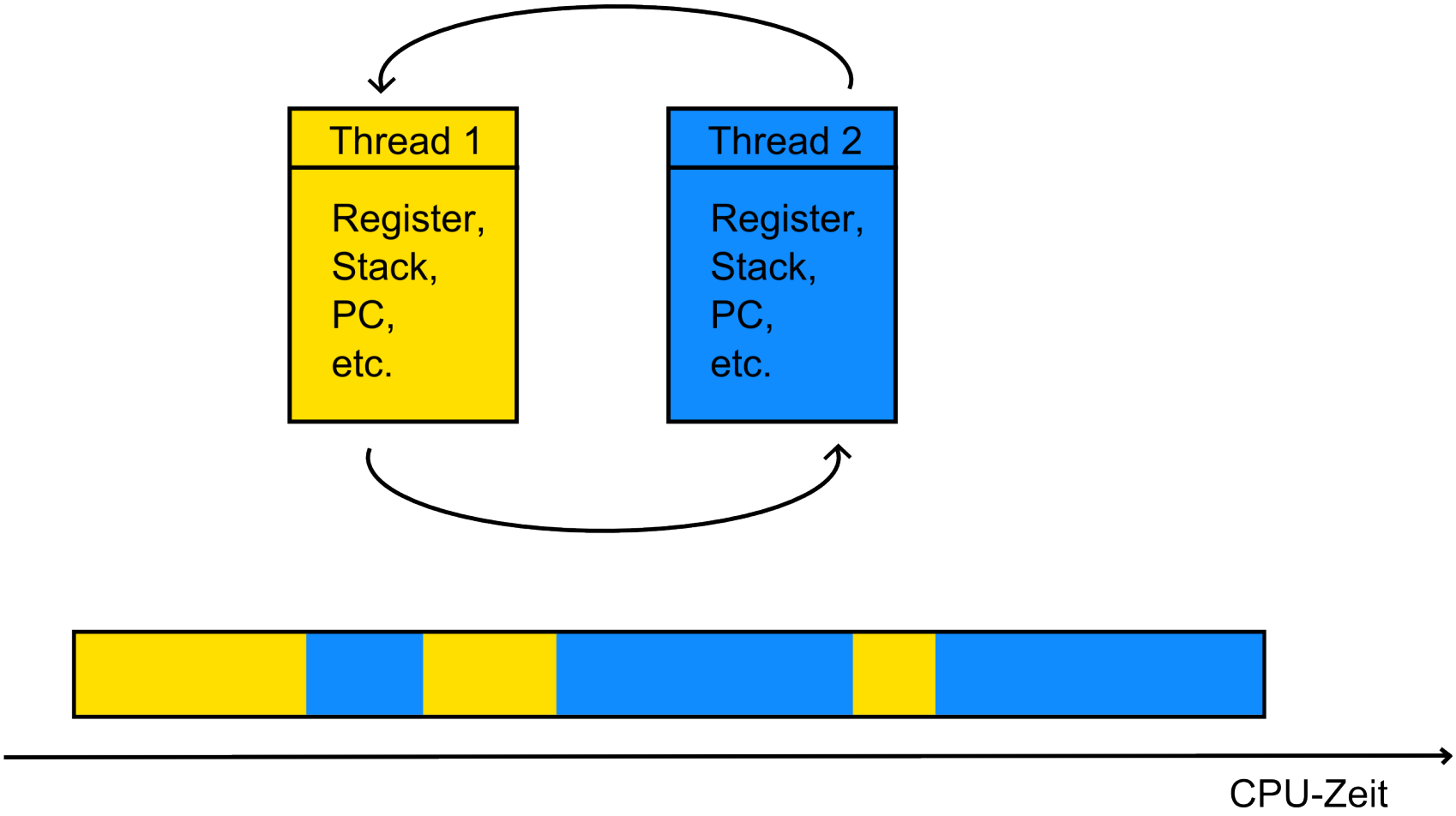
Threads

- “leichtgewichtige“ Prozesse
- kein eigener Heap, shared libraries, Text-Segment, etc.
- einem Prozess untergeordnet
- in demselben Adressraum wie der Prozess $\Rightarrow$ einfache Kommunikation
- ein Thread $\hat{\approx}$ eine Aufgabe, ein Prozess $\hat{\approx}$ eine Applikation



Überblick (2)

Context Switching

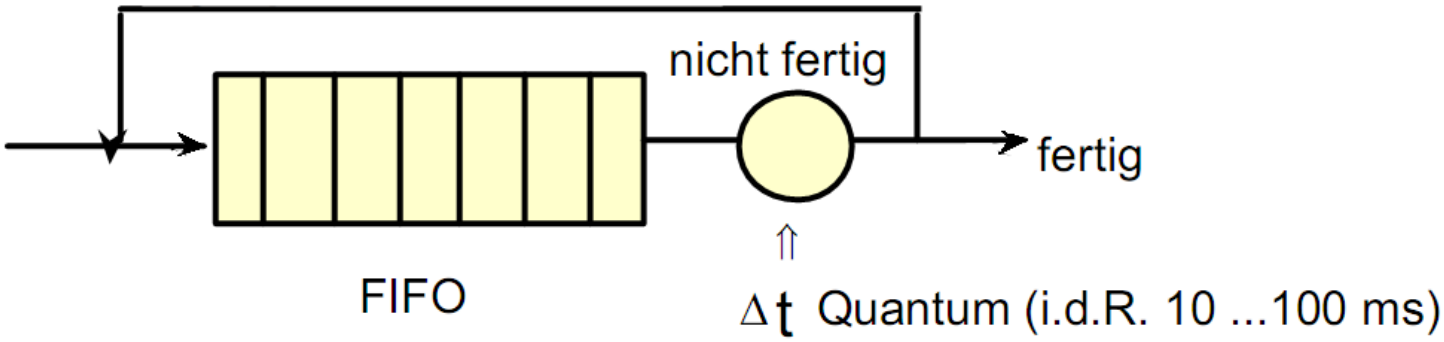


Überblick (3)

Scheduler

- bestimmt die Reihenfolge und Dauer der Context Switches
- Teil des Kernels
- Prioritäten

Round-Robin-Scheduler:

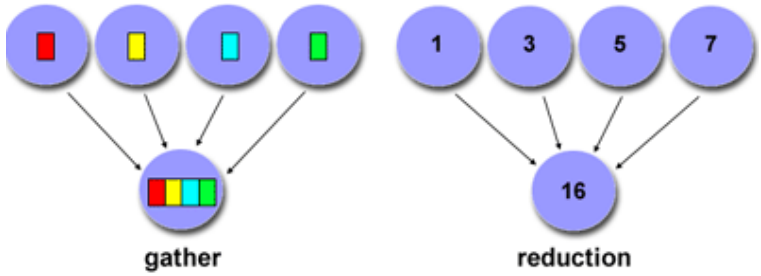


weitere Verfahren: FIFO, Prioritätsscheduling...

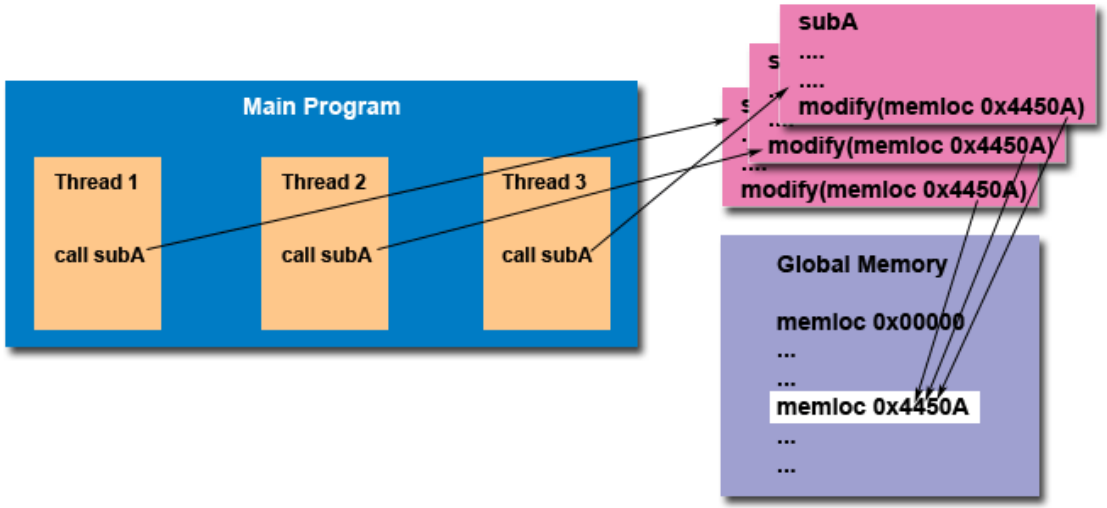
Überblick (4)

Synchronisation - warum?

Zusammenführen der parallelisierten Arbeit:



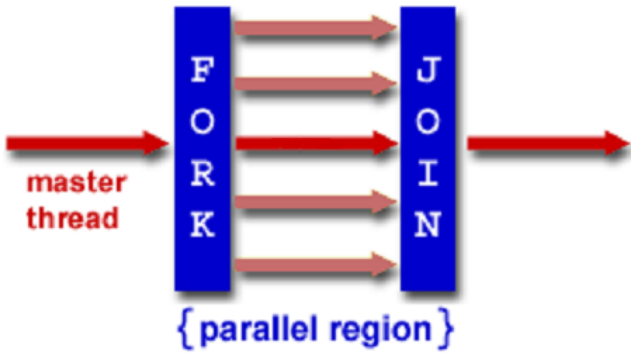
Problematik “Thread-safeness“:

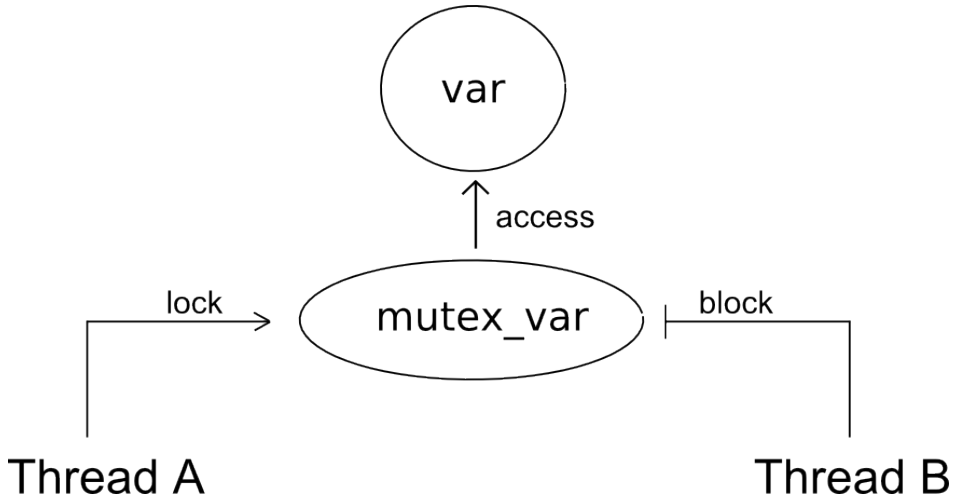


Überblick (5)

Synchronisation - Methodik

- Semaphoren + Mutexe
- condition variables
- Barrieren





Überblick (6)

Parallelisierbarkeit

Wann kann man parallelisieren?

- Aufteilung der Daten möglich (z.B. bei vielen Schleifen)
- Aufteilung nach Funktionalitäten mit schwachen Abhängigkeiten möglich (z.B. Modellierung des Ökosystems)
- Fibonacci-Reihe z.B. nicht parallelisierbar:
  $F(k+2) = F(k+1) + F(k)$

Ziel: optimale Auslastung der Threads (alle gleichzeitig fertig)

- durch statische, gleichmäßige Aufteilung der Arbeit
- durch dynamische Arbeitszuweisungen

Automatische Parallelisierung (1)

- Idee: serieller Code $\rightarrow$ paralleler Code
- auf verschiedenen Ebenen möglich (z.B. ILP)
- meist durch Einsatz spezieller Tools/Compiler
- hauptsächlich werden Schleifen analysiert und die Daten auf mehrere Threads aufgeteilt
- Vorteil: kein zusätzlicher Arbeitsaufwand
- Nachteile: nicht applikationsspezifisch, Compiler kann zu fehlerhaften Schlüssen kommen

Automatische Parallelisierung (2)

Beispiel: Instruction Level Parallelism

```
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
```

- i386-Code aus der Windows-Kernel-Dll “ntdll“
- Zeile 7: “interleaved code“ (Compiler-Optimierung)
- Voraussetzung: eine CPU mit mehreren ALUs (“superscalar CPU“)

Manuelle Parallelisierung (1)

Ebenen:

1. Programmierer: Effizienz, Thread-Safeness
2. OS auf dem eingebetteten System: Unterstützung von Prozessen, Threads, Scheduling
3. Hardware: mehrere Kerne, mehrere ALUs, “pipelined instructions“

- geschichtlich: jeder CPU-Hersteller stellt eine eigene hardware-spezifische API zur Verfügung
- heute: einige Standards vorhanden; z.B. POSIX Threads, OpenMP, MPI

Manuelle Parallelisierung (2)

POSIX Threads - Überblick

- IEEE-Standard 1003.1c aus dem Jahre 1995
- shared memory model
- als C-Bibliothek angedacht $\Rightarrow$ inzwischen Bestandteil der libc
- gcc -pthread ...
- Thread-Erstellung um Faktor 10 bis 100 schneller als bei Prozessen (fork())
- Thread-Kommunikation einfacher als Prozess-Kommunikation (shared memory)

Manuelle Parallelisierung (3)

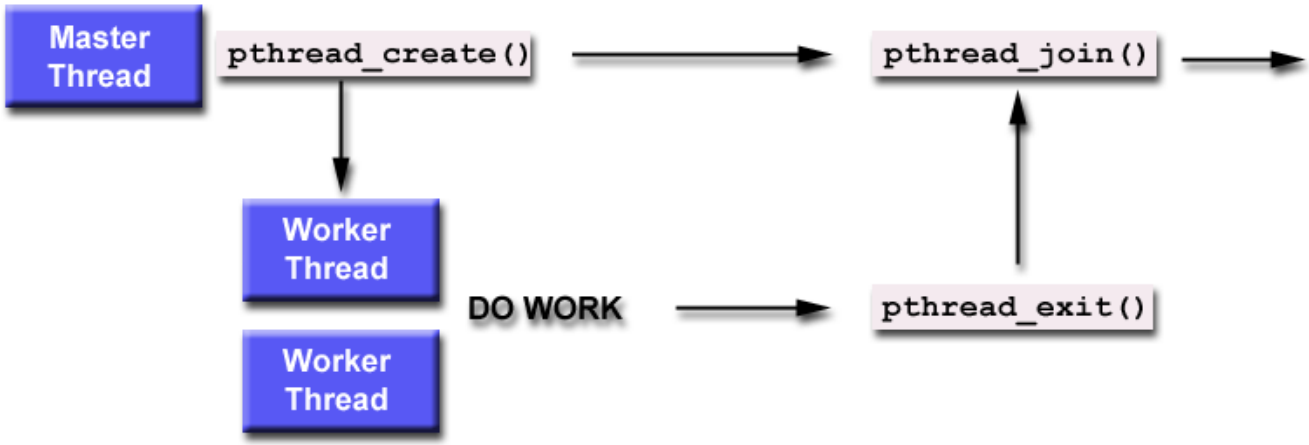
POSIX Threads - Beispiel

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #define NUM_THREADS     5
4
5  void *PrintHello(void *threadid)
6  {
7     long tid;
8     tid = (long)threadid;
9     printf("Hello World! It's me, thread #%ld!\n", tid);
10    pthread_exit(NULL);
11 }
12
13 int main (int argc, char *argv[])
14 {
15    pthread_t threads[NUM_THREADS];
16    int rc;
17    long t;
18    for (t = 0; t < NUM_THREADS; ++t) {
19       printf("In main: creating thread %ld\n", t);
20       rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
21       if (rc) {
22          printf("ERROR; return code from pthread_create() is %d\n", rc);
23          exit(-1);
24       }
25    }
26    pthread_exit(NULL);
27 }
```

Manuelle Parallelisierung (4)

POSIX Threads - Möglichkeiten

- Unterstützung diverser Synchronisationsmöglichkeiten: Semaphoren/Mutexe, condition variables, “Joining“



- optional: Setzen von Prioritäten, Festlegen von Scheduling Policies (“pthread_attr“)

Manuelle Parallelisierung (5)

OpenMP - Überblick

- 1997 von diversen Hardware- und Softwareherstellern (u.a. Intel, Sun, IBM, HP) definierte API $\Rightarrow$ Industriestandard
- für C/C++ und Fortran spezifiziert
- wird noch weiterentwickelt (Version 3.0 aus dem Jahre 2008)
- shared memory model
- “semi-manuelle Methode“
- Nutzung von Compiler-Direktiven (Compiler-Unterstützung benötigt: z.B. “gcc -fopenmp. . . “ ab gcc 4.3.2)

Manuelle Parallelisierung (6)

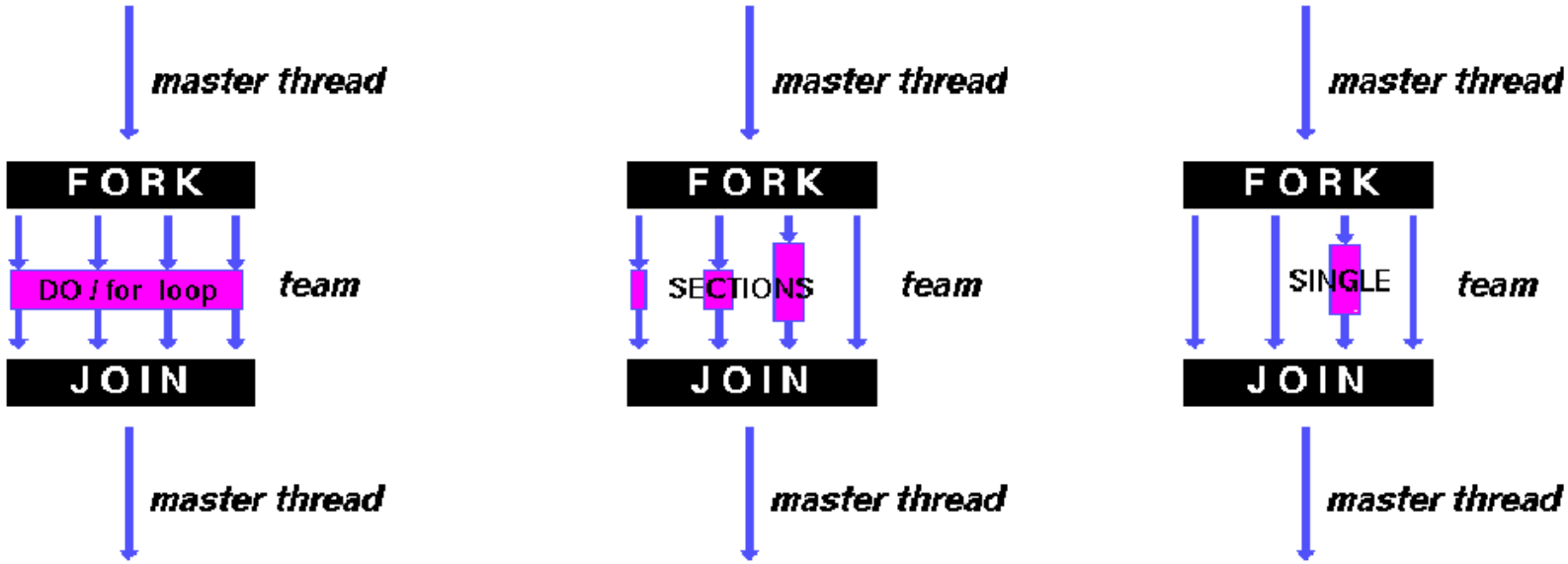
OpenMP - Beispiel

```
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 }
```

Manuelle Parallelisierung (7)

OpenMP - Zusammenfassung

- Fork-Join-Modell



- Vorteil: einfache Parallelisierung von “legacy code“, nicht unterstützende Compiler ignorieren die Direktiven schlichtweg
- Nachteil: Compiler erledigt die konkrete Parallelisierung $\Rightarrow$ Verlust der Kontrolle über einige Details

Manuelle Parallelisierung (8)

MPI

- MPI = Message Passing Interface
- Industriestandard aus den Jahren 1994 (MPI-1) und 1996 (MPI-2), $>$ 40 teilnehmende Organisationen
- distributed memory model (!)
- reiner Nachrichtenaustausch zwischen Threads, keine Threaderstellung
- Threaderstellung durch ein Programm (z.B. “mpirun“), meist ein Thread pro Prozessor
- häufig in hochperformanten Rechenclustern eingesetzt

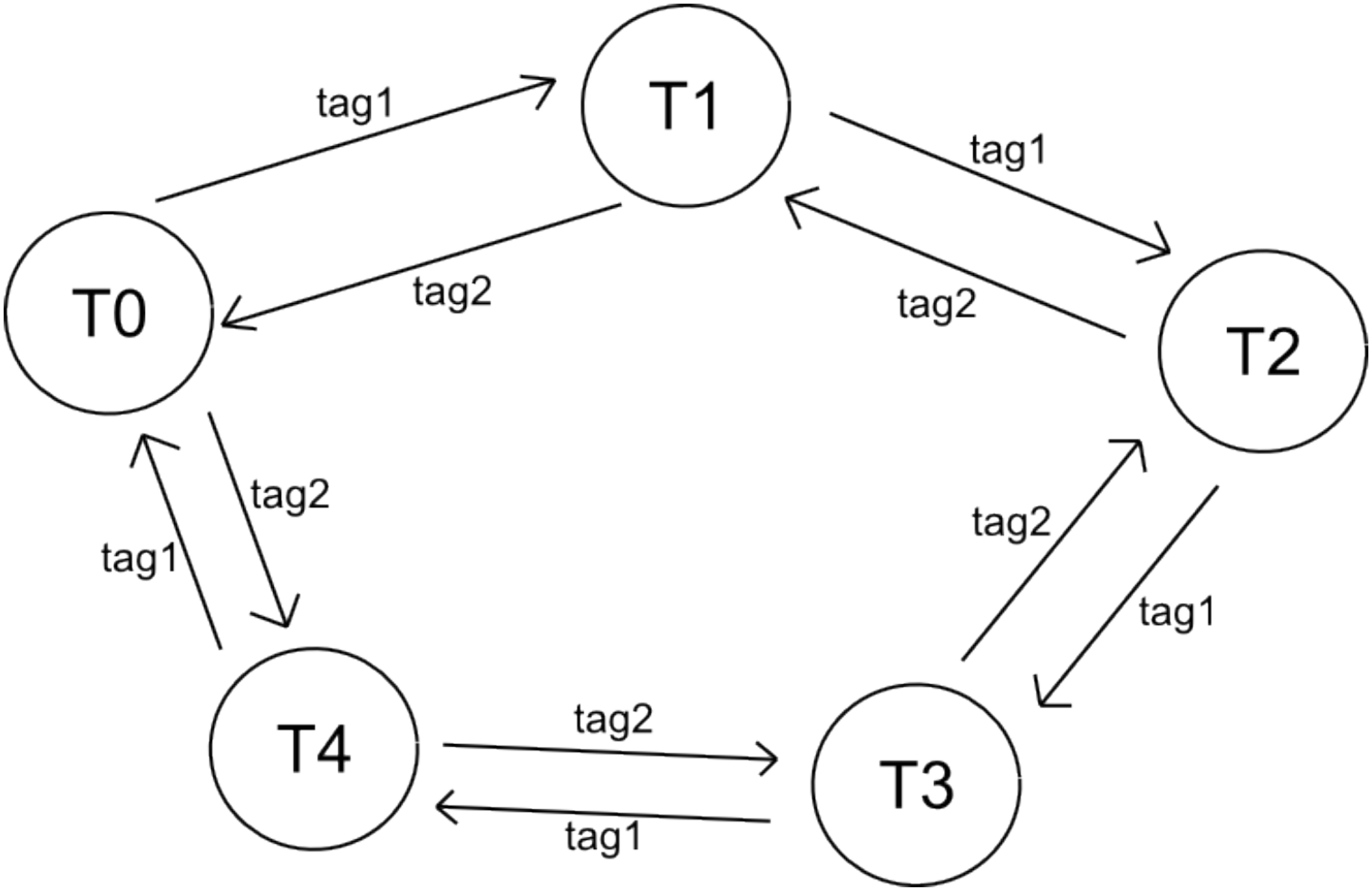
Manuelle Parallelisierung (9)

MPI - Beispiel

```
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 }
```

Manuelle Parallelisierung (10)

MPI - Ringtopologie



Echtzeitbedingungen (1)

Definition

- eine Aufgabe muss bis zu einem vorgegebenen Zeitpunkt erledigt sein, sonst ist ihr Ergebnis . . .
  - wertlos $\Rightarrow$ harte Echtzeitbedingungen (Flugzeug, Auto o.ä.)
  - weniger nützlich $\Rightarrow$ weiche Echtzeitbedingungen (z.B. Videostreaming)
- Unterstützung des OS nötig: Scheduler mit priorisierbaren Tasks, exakte Zeitmessungen, in der Zeit deterministische system calls $\Rightarrow$ Real-Time OS (RTOS): freeRTOS, ThreadX, QNX, LynxOS, Nucleus RTOS

Echtzeitbedingungen (2)

Scheduling

Prioritäts-Scheduling:

- Prozess mit höchster Priorität bekommt die CPU
- zwischen Prozessen gleicher Priorität wird ein RR-Scheduling durchgeführt

wie werden die Prioritäten vergeben?
(Ann.: periodisch wiederkehrende Aufgaben)

- statisch $\Rightarrow$ Rate Monotonic Algorithm optimal

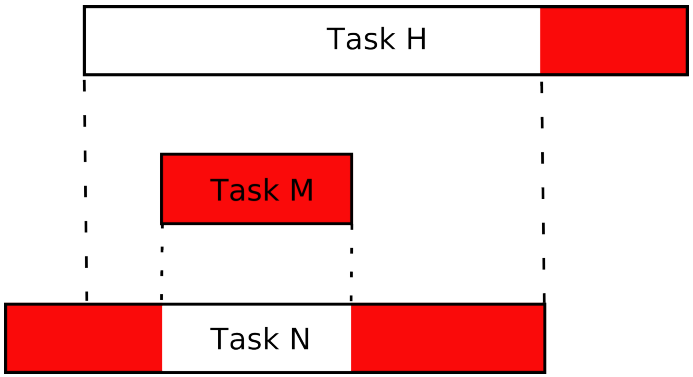
Rate Monotonic Algorithm

Die Aufgabe mit der kleinsten Periode erhält die größte Priorität.

- dynamisch $\Rightarrow$ deutlich komplexer, aber meist bessere CPU-Auslastung erreichbar (z.B. Priorität nach Ausführungszeit verringern)

Echtzeitbedingungen (3)

Problem: Prioritäts-Inversion



Lösungen:

- Vererbung der Priorität von Task H an Task N
- Vergabe einer Priorität m an die Resource, inne habender Task erhält Priorität $m + 1$
- “thread migration“ von Task N bei multicore-Architekturen

Zusammenfassung

- Voraussetzung: Partitionierung des Problems möglich
- Beschreibung paralleler Systeme mit bisherigen Standards scheint ausreichend
- Echtzeitbedingungen als besondere Schwierigkeit

Trends:

- flexiblere Scheduler: Energieverbrauch berücksichtigen (bislang: Nutzung der CPU, des Speichers, der I/O), sich zur Laufzeit ändernde Anforderungen an Tasks berücksichtigen (z.B. Umwelteinflüsse)
- jedem Task einen festen Anteil an der CPU-Zeit zusichern

Literatur


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


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


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


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


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


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


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


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


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


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

Fragen

Fragen?