Eingebettete Systeme

Echtzeitverhalten und Betriebssysteme

1. Einführung

Example software development

Target system

- Application tasks
- Real-time OS (vxWork)
- Hardware (MBX860)

Development host

- Compiler, debugger, loader, simulator, shell monitor
- NT OS (Solaris)
- Hardware (MBX860)
- Pentium PC station
- SUN workstation

Input/output

RS-232 Ethernet

Software development

Binary Code

OS

Debugger

Compiler

Software Source Code

Simulator Debugger

USER

Trends

- Embedded CPU-Cores are becoming smaller and more powerful
- Higher computing power because of higher clock rate, deeper pipelines, branch prediction, more dynamic decisions, caches, ...
- Deeper integration of CPU-cores with
  - programmable hardware
  - (analog) interfaces
  - digital signal processors
  - memory

1989

1993

1995

1999

> 500k transistors
1 - 0.8 µm
33 MHz

2+M transistors
0.8 - 0.65 µm
75 - 100 MHz

5+M transistors
0.6 - 0.35 µm
133 - 167 MHz

22+M transistors
0.25 - 0.18 µm
500 - 600 MHz
**Trends**

- Memory becomes a critical resource

- Communication plays an important role in embedded systems
  - Distributed objects DCOM CORBA,
  - Firewire,
  - Bluetooth,
  - USB,
  - IrDA,
  - Jini,
  - TCP/IP,
  - …

**Consequences**

- Critical components of a software development environment:
  - Compiler (efficient code generation, predictable execution times of tasks and processes)
  - Interfaces ((automatic) generation of device drivers for external units and between hardware and software components)
  - Embedded operating systems (memory management, process management, I/O, protection mechanisms, synchronization, communication)

**Interfaces**

What does a device driver do?

- Initialization
  - Register with operating system
  - Initialize interrupt service
  - Allocate memory
  - Initialize hardware

- Data access
  - Data exchange between application and hardware (e.g., memory mapped, DMA)

- Processing of protocol layers

- Interrupt service
Interfaces

- **Automatic software generation** is often part of software development environments, e.g. Aisys DriveWay, Stenkil MakeApp, Intel ApBuilder, Motorola MCUnit, CoWare, ...
  - properties of devices are usually stored in a database; software generation has access to this database
  - specific drivers between hardware and software components are synthesized

Compiler

- Because of the **complexity** of computer architectures and applications, we find more and more software written in high level languages like C++, Java, Ada.
- Because of **cost constraints** (just a few installations, many different processor families exist), the compiler support for embedded systems is rather poor (RedHat Cygnus, TI Tartan, Green Hills). Demand for retargetable compilers is growing.
  - The compilation quality for embedded microprocessors (e.g. ARM, MIPS, PowerPC, ...) is satisfying whereas the code quality for DSPs is rather poor.
    - specialized data paths, e.g. multiply-accumulate
    - multiple memory banks and busses
    - specialized addressing modes, e.g. circular buffers
    - specialized instruction sets, e.g. zero overhead loops
    - specialized peripheral units

  - **Important criteria**
    - Short and predictable execution times of processes
    - efficient use of memory structure
    - small code size
Compiler

Operating systems (OS)

- Software layer between application and hardware.
- Major task of an OS is the dynamic management of memory, communication and computing resources:
  - Memory management (main memory, secondary memory, virtual memory, memory protection, paging/segmentation, file system)
  - Process management (scheduling, task queuing, synchronization, communication)
  - Security and protection mechanisms
  - I/O management (device driver, scheduling, networking)
  - Support for distributed hardware resources and distributed applications
- Desirable features: Timeliness, design for peak load, predictability, fault tolerance, maintainability

Why multiple processes?

- The concept of concurrent processes reflects the intuition about the functionality of embedded systems.
- Processes help us manage timing complexity:
  - multiple rates
    - multimedia
    - automotive
  - asynchronous input
    - user interfaces
    - communication systems

Example: engine control

- Tasks and processes:
  - spark control
  - crankshaft sensing
  - fuel/air mixture
  - oxygen sensor
  - Kalman filter – control algorithm
A first concept: co-routines

- Programming technique commonly used in the early days of embedded systems.
- Example of devise complex control without using processes.
- Like subroutine, but caller determines the return address.
- Co-routines voluntarily give up control to other co-routines.
- Pattern of control transfers is embedded in the code.

Problems:
- difficult to determine execution trace from program
- no information hiding

Co-routine methodology

Example ARM assembler:

- co-routine 1
  - ADR r14, co2a
  - co1a: ...
  - ADR r13, co1b
  - MOV r15, r14
  - co1b: ...
  - ADR r13, co1c
  - MOV r15, r14
  - co1c: ...

- co-routine 2
  - ADR r14, co2b
  - co2a: ...
  - MOV r15, r13
  - co2b: ...

- r15 is the PC

Processes

- A process is a unique execution of a program.
  - Several copies of a program may run simultaneously or at different times.

- A process has its own state:
  - registers;
  - memory.

- The operating system manages processes.

Processes and CPUs

- Activation record:
  - copy of process state
  - includes registers and local data structures

- Context switch:
  - current CPU context goes out
  - new CPU context goes in
Co-operative multitasking

- Each process allows a context switch at `cswitch()` call.
- Separate scheduler chooses which process runs next.

- Advantages:
  - hides context switching mechanism
  - relies on processes to give up CPU
  - predictable, where context switches can occur.

- Problems:
  - programming errors can keep other processes out, process never gives up CPU
  - real-time behavior at risk if it takes too long before context switch allowed

Example: co-operative multitasking

```c
Process 1
if (x > 2)
    sub1(y);
else
    sub2(y);
cswitch();
proca(a,b,c);
```

```c
Process 2
procdata(r,s,t);
cswitch();
if (val1 == 3)
    abc(val2);
rst(val3);
```

Preemptive multitasking

- Most powerful form of multitasking:
  - OS controls when contexts switches;
  - OS determines what process runs next.

- Use timer to call OS and switch contexts:

- Use hardware or software interrupts, or direct calls to OS routines to switch context

Flow of control with preemption
Process management

Important tasks of process management:

- Execution of quasi-parallel tasks on a processor using processes or threads (lightweight process) by
  - maintaining process states,
  - process queuing,
  - preemptive tasks (fast context switching) and quick interrupt handling
  - dealing with dynamically generated and terminated processes
- CPU scheduling (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
- Process synchronization (critical sections, semaphores, monitors, mutual exclusion)
- Inter-process communication (buffering)
- Support of a real-time clock as an internal time reference
- Common use of resources

Minimal set of process states: