University of Glasgow | School of Computing Science

Programming Real-time and Embedded Systems

Advanced Operating Systems (M)
Lecture 10

Lecture Outline

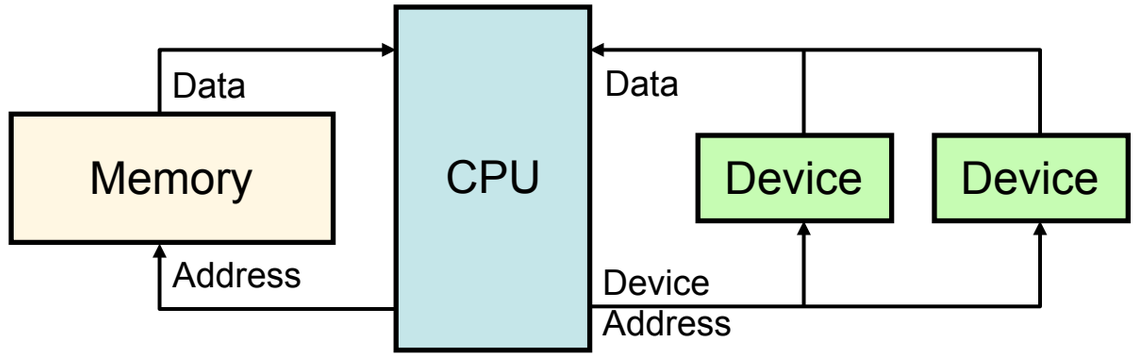
- Programming real-time and embedded systems
  - Interacting with hardware
  - Interrupt and timer latency
  - Memory issues
  - Power, size and performance constraints
  - System longevity
  - Development and debugging

Real-time and Embedded Programming

- Real time and embedded systems differ from conventional desktop applications
  - Must respect timing constraints – scheduling theory in prior lectures
  - Must interact with hardware and the environment
  - Often very sensitive to correctness and robust operation
  - Often very sensitive to cost, weight, or power consumption
- Implications to consider:
  - Proofs of correctness, scheduling tests, etc.
  - Limited resources: low level programming environments; high awareness of systems issues; interaction with hardware
  - Challenges imposed on operating system and programming environment by resource constraints and programming model

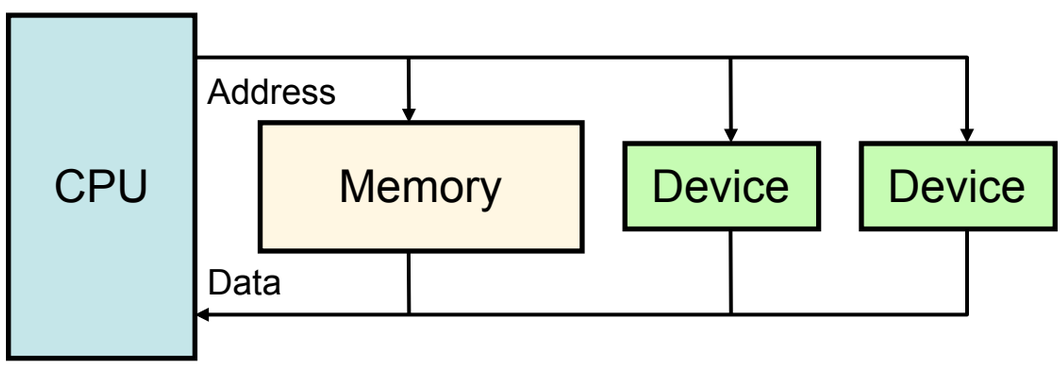
Interacting with Hardware: Concepts

- Separate device bus:
  - Different assembler instructions to access devices than to access main memory
  - Separate physical connection to the device address/data bus
  - Example: early Intel PC hardware; kept for backwards compatibility on modern PCs, but rarely used
  - Program using inline assembler



Separate device bus

- Memory mapped I/O:
  - Devices appear at some address in memory; access as-if any other part of system memory
  - Single set of assembler instructions for memory access; single address and data bus on processor
  - Example: most modern systems
  - Program using high level language

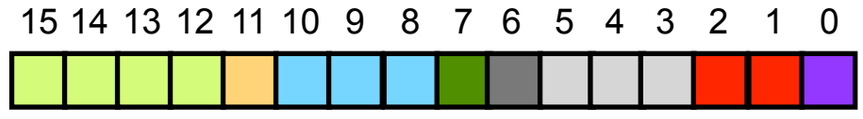


Memory mapped architecture

Interacting with Hardware: Concepts

- Two models for device control:
  - Polling: device sets bit in control register to indicate an event occurred; software periodically inspects that register, decides whether to take some action
  - Interrupt driven: device sets bit in control register to indicate some action occurred, then asserts a processor interrupt to notify software; software responds to interrupt and takes appropriate action
- Most current hardware is interrupt driven, but can usually be switched to polled mode (useful for high rate sources to avoid interrupt load, or if running a cyclic executive

- Control registers presented as bit fields at some address range
  - Bit-level manipulation needed to access fields in control register
  - Need to know memory address and size of control register, layout, endianness, and meaning of various bit fields within the control register,
- Some bits may be read- or write- only; some may change when read




```
Bits
15 - 12  : Errors
11       : Busy
10 -  8  : Unit select
 7       : Done/ready
 6       : Interrupt enable
 5 -  3  : Reserved
 2 -  1  : Device function
 0       : Device enable
```

Interacting with Hardware: C

```
struct {
    short errors     : 4;
    short busy       : 1;
    short unit_sel   : 3
    short done       : 1;
    short irq_enable : 1
    short reserved   : 3
    short dev_func   : 2;
    short dev_enable : 1;
} ctrl_reg;

int enable_irq(void)
{
    ctrl_reg *r = 0x80000024;
    ctrl_reg  tmp;

    tmp = *r;
    if (tmp.busy == 0) {
        tmp.irq_enable = 1;
        *r = tmp;
        return 1;
    }
    return 0;
}
```

- C allows definition of bit fields and explicit access to a particular memory address using pointers
  - Example on left shows simple manipulation of a control word at address 0x80000024
- Allows implementation of device drivers and interrupt handlers
- Illusion of portable code: standard C does not specify:
  - Size of basic types (e.g., number of bits in a byte, number of bytes in an int)
  - Bit and byte ordering
  - Alignment or atomicity of memory access
  - Each compiler/operating system platform defines these appropriately for its environment

Interacting with Hardware: Ada

```
type ErrorType   is range 0..15;
type UnitSelType is range 0..7;
type ResType     is range 0..7;
type DevFunc     is range 0..3;
type Flag        is (Set, NotSet);
type ControlRegister is
record
    errors    : ErrorType;
    busy      : Flag;
    unitSel   : UnitSelType;
    done      : Flag;
    irqEnable : Flag;
    reserved  : ResType;
    devFunc   : DevFunc;
    devEnable : Flag;
end record;

for ControlRegister use
record
    errors    at 0*Word range 12..15;
    busy      at 0*Word range 11..11;
    unitSel   at 0*Word range  8..10;
    done      at 0*Word range  7.. 7;
    irqEnable at 0*Word range  6.. 6;
    reserved  at 0*Word range  3.. 5;
    devFunc   at 0*word range  1.. 2;
    devEnable at 0*Word range  0.. 0;
end record;

for ControlRegister'Size use 16;
for ControlRegister'Alignment use Word;
for ControlRegister'Bit_order use Low_Order_First;
...
```

- Ada has extensive support for low-level hardware access and interrupt handlers
  - Precise control over record layout in memory, byte ordering, bit size of types, etc.
  - Perhaps overly verbose...?
  - Facilities for interrupt handlers in the language
- Allows portable code to be written that manipulates hardware devices

Language Support for Hardware Access

- Language support for portable hardware access conceptually nice, but less useful than might be expected
  - Real time embedded systems typically tied to platform due to specialist hardware
  - Little need for portability at the language level, since underlying system unique
- Main advantage of Ada: strong type checking for hardware access

Interrupt and Timed Task Latency

- Devices typically request service using interrupts
  - Need predictable worst-case bounds on service time, otherwise cannot reason about the system
  - Both interrupt latency and task scheduling latency
- Examples:
  - Linux has ~600µs typical interrupt handler latency, often runs with 100Hz clock for task scheduling (i.e. 10000µs latency)
    - Long history of problems with system call latency, causing tasks to block for hundreds of milliseconds on certain device accesses
    - Resolved for most common devices, but still unpredictable (and long) latency with uncommon hardware
  - RTLinux claims a maximum 15µs interrupt handler latency, all scheduled tasks execute within at most 35µs of their scheduled time
    - Other hard real-time operating systems offer similar guarantees

Interrupt and Timed Task Latency

- Why such a difference?
  - Preemptable microkernel, with single address space
    - No context switch, user-to-kernel mode, overhead
  - No virtual memory or memory protection
    - No paging delays
    - No delays while page tables adjusted
  - Device drivers designed with minimal non-preemptable sections
    - Light-weight, prioritised, threads fire in response to interrupts

- Does it matter? It depends on the application…

Memory Protection

- Many embedded systems use a single flat address space
  - Applications, shared libraries, kernel, devices all visible
  - A system or library call is equivalent to a function call

  - Makes system calls, interrupts, very fast and predictable: no context switch to kernel mode; no adjustment of MMU page tables
  - Consequences: no isolation between applications, or between applications and the kernel
  - A change to one part implies that the entire system has to be revalidated; difficult as systems become larger
  - Some systems offer limited protection: read only mapping of program/ system text; IRQ vectors

Memory Protection

- Consequences of offering memory protection:
  - Unpredictable latency
    - May take longer to task switch to/from a protected task
  - Memory overhead
    - Protection provided on a per-page basis, leads to wastage
    - Overhead of maintaining the page tables and protection maps
  - Code overhead
    - Operating system is required to trap illegal access and recover system to a safe state

- Which is easiest: proving system correct, or writing handlers to safely recover from all possible errors?

Virtual Memory: Address Translation

- Address translation: act of making a fragmented block of physical memory appear to be a single contiguous block
  - Useful in dynamic systems: enables requests for large blocks of memory to be allocated when there is no physically contiguous block available
  - Adds overhead, since system must manage address translation tables
    - Uses memory, increases context switch time
    - Complicates DMA device access
- Better to pre-allocate static memory pools for real-time tasks
  - Manage the sub-division of address space within the application

Virtual Memory: Paging to Disk

- Disk based virtual memory is supported by many systems that run both real time and non-real time tasks
  - Paging to disk clearly impact real-time performance
  - Unpredictable delays, depending whether page is in memory or on disk
- Systems usually provide ability to (selectively) prevent paging
  - POSIX allows regions of memory to be locked into RAM and preventing from paging using `mlock(addr, len)` and `mlockall()`
  - Windows allows all memory owned by a particular thread to be locked

Memory Leaks and Garbage Collection

- An embedded system has to run for a long period of time, without user intervention
  - C programs typically have memory leaks due to programmer error
    - Significant problem in long-lived or resource constrained systems
    - Better to pre-allocate static buffers, avoid the chance of a memory leak
    - Be very careful to free memory and other resources after use
    - Do you always check for out of memory errors and recover gracefully? Recall: the recovery code cannot allocate memory (this may include the stack frame needed to make a function call!)
  - Modern languages use garbage collection to avoid resource leaks
    - Has a poor reputation due to unpredictable delays when collection occurs, but real-time garbage collection algorithms – with predictable latency, at controlled times – do exist

Memory: What is a Small System?

- Embedded systems often very constrained compared to typical desktop computers
  - You may be running on an 8 bit processor, with kilobytes of RAM
  - Operating system typically optimised for the environment, provides only minimal required functions
    - The QNX 4.x microkernel is approximately 12kbytes in size
    - The VRTX microkernel is typically 4-8kbytes in size

- For comparison:

```
# uname -srm
Darwin 10.5.0 i386          (MacOS X 10.6.5)
# cat tst.c
int main()
{
    return 0;
}
# gcc tst.c -o tst
# ls -l tst
-rwxrwx---  1 csp  staff  8664 29 Dec 15:24 tst
```

Effects of Cache

- You may be running on a more modern processor
  - PowerPC 405CR embedded processor
    - 32 bit RISC processor, compatible with desktop PowerPC
    - 133MHz or 266MHz clock speed
    - 500mW power consumption: Intel Atom (NetBook processors) consume ~2.5W
    - CodePack™ compression of executables
    - Likely has several megabytes of memory
  - Relatively cheap, comparatively high performance, low power
- Has a small cache, which you may want to disable:
  - Processor and memory speeds are closely matched
    - Compare to desktop processor, with order magnitude difference
  - Simpler to predict memory access times without the cache
  - Cache improves average response times, but introduces unpredictability

Power, Size and Performance Constraints

- Embedded systems often constrained hardware
  - May have limits on power consumption (e.g., battery powered)
  - May have to be physically small and/or robust
  - May have strict heat production limits
  - May have strict cost constraints
    - That processor is slower, but 10¢ cheaper, the production run is 1 million, you paid your salary for the next couple of years…
- Used to throwing hardware at a problem, writing inefficient – but easy to implement – software
  - Software engineering based around programmer productivity
  - The constraints may be different in the embedded world…

System Longevity

- Systems may be safety critical or difficult to update
  - e.g., medical devices, automotive or flight control, industrial machinery
  - e.g., CD or DVD player, washing machine, microwave oven
- May need to run for many years, in environment where failures either cause injury or are expensive to fix
  - Can you guarantee your system will run for 10 years without crashing?
  - Do you check all the return codes and handle all errors?
  - Fail gracefully?

Development and Debugging

- Systems may be too limited to run compiler
  - Develop using a cross compiler running on a PC, download code using a serial line, or by burning a flash ROM and installing
- May have limited debugging facilities:
  - Serial line connection to host PC
  - LEDs on the development board
  - Logic analyser or other hardware test equipment
  - Formal proofs of correctness become more attractive when real system so difficult to analyse…

Summary

- Low level and embedded programming
  - Control register access, bit packing, alignment, etc.
  - Interrupts
  - Memory
    - Address translation
    - Paging and virtual memory
    - Allocation and garbage collection
    - Caching
- Consider:
  - Systems issues, how features that improve general purpose performance hinder real time systems
  - Constraints on embedded systems, differences in engineering compared with general purpose systems