Overview

• problems with busy-wait
• reactive objects view
• OO programming model and reactive objects

Recapitulation

• Our achievements so far:
• We know how to read and write to I/O device registers
• We know how to run several computations in "parallel" by time-slicing the CPU
• We know how to protect critical sections by means of a mutex
• But...

Recapitulation

• We still haven’t thoroughly addressed the problem of I/O synchronization!
• Each thread – half the CPU cycles
  • even if unused
• CPU usage – computations – wasted!
  • busy-wait

Consequence 1

• Long wait for status change
  – With N threads in the system
  – delay will be T*(N-1) in the worst case
    • all other threads executed before

Consequence 2

• Busy-waiting
  – waiting indistinguishable from computing!
• In example
  – would have finished in time
    • if B hadn’t executed
Conclusion

- Time-slicing and busy-waiting
  - not a good technique for efficient input synchronization
  - in a system that must meet real-time constraints
- Add to that:
  - 100% CPU cycle usage = 100% power usage = short battery lifetimes (where applicable)
- need input synchronization technique
  - that doesn’t require the receiver of data to actively ask whether it is there
- Let’s rewind the tape a bit...

Recall reactive model

- Notice irregularity
  - cpu initiates both...
    - read
    - write
  - problem to know when to read
    - busy-wait
      - (so far)

More appealing view

- Port initiated write!
  - why can’t the HW tell cpu when data is present?
    - it can!
- how?

Interrupts

- On hardware level
  - interrupt signals
    - can be seen as port-initiated write operations
  - interrupts
    - need for busy-waiting disappears
  - compare to:
    - checking into the post-office again and again to see if a delivery has arrived, or
    - receiving a note in your mailbox that the goods can be picked up
- The CPU reacts to an interrupt signal by executing a designated interrupt service routine (ISR)
  - cpu receives signal
  - immediately starts executing ISR
  - regardless of current code executing
  - later resumes executing regular code

ISR consequences

- thread stopped (no context switch)
  - ISR started
    - automatically
    - on interrupt
- ISR performs necessary work
  - keep ISR small
    - no concurrency when executing ISR
  - often communicates with thread
    - thread start performing something else
- If thread may not be interrupted
  - turn interrupts off
    - for a short while

ISRs

- Encode the way a CPU reacts to events in its environment
  - events generated by port devices connected to the external world
- C.f. how port devices might start some internal logic sequence in response to a command register write
- ISR
  - function whose address stored in predefined memory cell
  - (23 such cells for 23 different external signals on the ATmega169)
- compared to traditional algorithmic programming and read/write I/O
  - ISR represent a conceptually “inside-out” model
    - external input directly executes code!
A CPU seen from the outside

- The program
  - code fragments
  - reactions to recognized events
- Input detection
  - invocation of code fragment
- CPU performs actions based on outside events
  - actions are code

A CPU seen from the inside

- The program
  - thread of control
  - run from start to stop (or infinitely)
  - read and write data on the way
- Input detection
  - exit from busy-wait loop
- Single program

Decision time

- internal, active view
  - possibly combine it with concurrent threads
  - hybrid between the active and the external (reactive) view (+ threads)
  - threads with interrupts
  - Most real-time literature and programming courses do, with an emphasis on the operating system that reconciles the views. However...
- we could explore the purely reactive view!
  - reactive objects

The reactive embedded system

- application
  - HW drivers (knows how to interact with HW)
  - software logic
    - performs desired computation
- The typical embedded system

- application
  - OS
  - driver
  - port
  - abstract hardware I/O
  - reactive interrupt model

- software reacts to events
  - computes result
  - write to port
Reactive objects

- Each box, be it a software or hardware component, represents a reactive object which
  - Maintains an internal state (variables, registers, etc)
  - Provides a set of methods as reactions to external events (ISRs, digital sequencers, etc)
  - Simply rests between reactions
  - May react in parallel with other objects (threads, chips)
- Each arrow represents an event/signal/message flow between objects, which can be either
  - asynchronous (write, sender continues in parallel)
  - synchronous (read, sender waits for a reply)

Example

- Internal parameter variables used by control algorithm
- Message = events = command to HW registers
- Physical object
- Message = events = inter-thread data transfer
- Internal parameter variables used during decode
- Message = events = interrupts from hardware

Reactive objects

- We have rediscovered the classic object-oriented programming model
  - Objects have local state
  - Objects export methods
  - Objects communicate by sending messages
  - Objects rest between method invocations
- Compare to
  - Intuitive object systems from the classic OO literature: people, cars, molecules, ...
- As a big bonus
  - Object-oriented design principles
  - Become applicable to embedded, event-driven & concurrent systems!

OO languages

- We cannot rely on the features of our common object-oriented languages :-(
- Because:
  - Creating a traditional object just means allocating a passive piece of storage, not creating a new thread of control
  - Object state is not local in the concurrency sense — being heap-allocated, it is implicitly shared by all threads
  - Method calls just mean calling subroutines, no concurrency or synchronization is involved

OO & concurrency

- Our model
  - Independent process
    - Local state
    - Method
    - Process fragment
    - Sending message
    - Inter-process communication

- Reactive objects view
  - Method invocation (message pass)
    - Blocks the caller
    - Process message in thread of called object
    - Return message
    - This is often how we actually think
  - Performs the same thing
    - What if we do NOT block the caller?
    - Continues executing
    - Does not work in OO language

OO Language consequences

- Object is
  - Passive piece of global state
- Method is
  - Subroutine
- Sending message
  - Subroutine call

- Object Languages
  - Object is
    - Passive piece of global state
  - Method is
    - Subroutine
  - Sending message
    - Subroutine call
Reactive objects by hand

- Concretely, we need to establish ways to
  - create reactive objects (define constructors)
  - declare protected local state (instance variables)
  - receive messages (define methods)
  - send messages (make method calls)
    - synchronously (sender waits)
    - asynchronously (sender continues)
  - bridge the hardware/software divide (run ISRs)
  - schedule a system of reactive software objects
- Next lecture will address each of these issues in turn