Invited presentation at Fujitsu Laboratories Workshop on Cyber-Physical Systems, Sunnyvale, CA, June 10, 2010.

Other workshop presenters included Bjarne Stroustrup (creator of C++), Jeannette Wing (National Science Foundation and Carnegie Mellon), and Edward Lee (U.C. Berkeley researcher on embedded real-time systems).

http://www.kellytechnologygroup.com/main/fujitsu-cps-workshop-10jun2010-invited-presentation.pdf

A paper based on a portion of this presentation can be found at:

http://www.kellytechnologygroup.com/main/concurrent-embedded-systems-website.pdf

David M. Cummings, Kelly Technology Group

Issues and Experiences in Developing Networked, Multithreaded Embedded Systems

David M. Cummings
Kelly Technology Group
June 10, 2010

Outline

- Embedded systems background
- Common requirements / architectural features
- Thread structure and thread communication
- Layering
- Conclusions

Embedded Systems Background

- 30 years designing and building embedded systems and appliances
- Most are highly multithreaded with significant networking / communication
- Examples:
  - Network of 27 Doppler radar processors for FAA's air traffic control system
  - Digital receiver for NASA's Deep Space Network
  - Flight computer for NASA's Mars Pathfinder spacecraft
  - TDMA satellite modem card for broadband IP access over the air
  - Terrestrial wireless appliance for broadband IP access over the air
  - Fibre Channel switches for Storage Area Networking
  - Clustered backup appliance

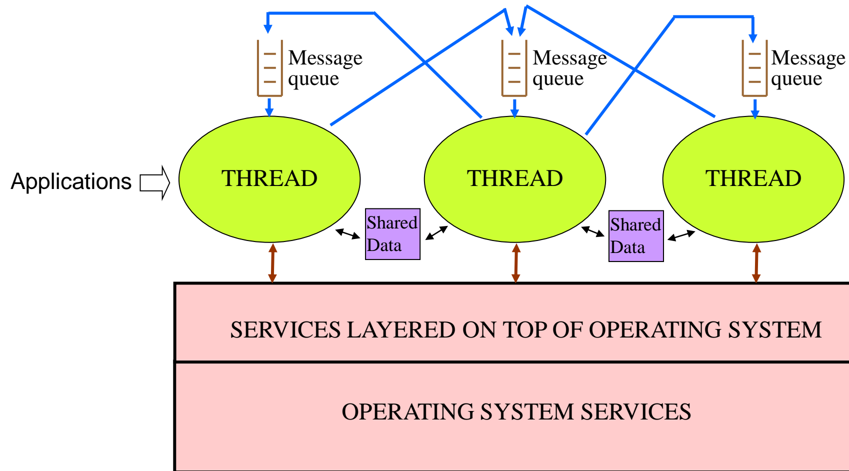
Recurring Requirements and Usage Issues

- Real-time deadlines
- Extensive interaction with hardware
- Largely or completely autonomous
- Often thought of as hardware devices that "just work"
  - Highly dependable
- Extensible
- Maintainable

Common Architectural Features

- Running at least a minimal real-time kernel
  - Threads
  - Preemptive, priority-based scheduling
  - Message passing
  - Synchronization primitives
- Developers think in terms of threads
- Written in C or C/C++ (one written in Ada 83)
  - No significant use of C++ objects

June 10, 2010 David M. Cummings, Kelly Technology Group 5

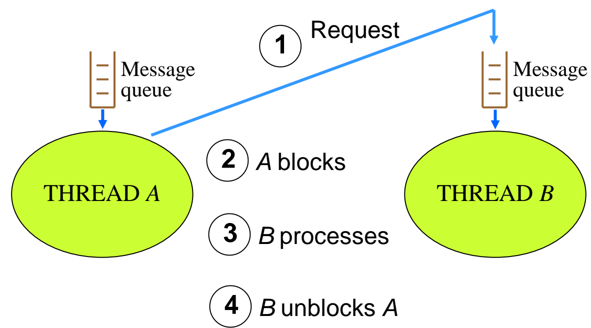

Typical Threading Model
Message queue
Message queue
Message queue
Applications
THREAD
THREAD
THREAD
Shared Data
Shared Data
SERVICES LAYERED ON TOP OF OPERATING SYSTEM
OPERATING SYSTEM SERVICES
June 10, 2010
David M. Cummings, Kelly Technology Group
6

Typical Thread Structure

```
INITIALIZE
DO Forever
    Wait for a message
    CASE message ID
        Msg 1: Process Msg 1
               Break
        Msg 2: Process Msg 2
               Break
        Msg 3: Process Msg 3
               Break
    END CASE
END DO
```

Example Communication Approach

- Request-Response



June 10, 2010 David M. Cummings, Kelly Technology Group 8

Request-Response Implementation

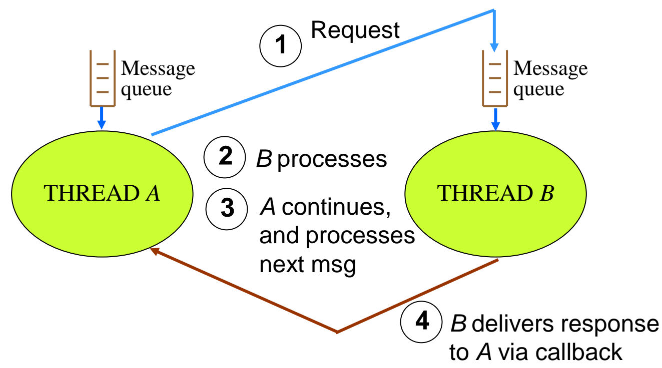
Thread *A*:

```
INITIALIZE
DO Forever
    Wait for a message
    CASE message ID
        Msg 1: Process message
               Send request to Thread B
               Wait for response
               Process response
               Break
        Msg 2: Process Msg 2
               Break
        Msg 3: Process Msg 3
               Break
    END CASE
END DO
```

June 10, 2010 *David M. Cummings, Kelly Technology Group* *9*

Example Communication Approach

- Callback



Callback Implementation

Thread *A*:

```
INITIALIZE
DO Forever
    Wait for a message
    CASE message ID
        Msg 1: Process message
               Send request to Thread B
               Break
        Msg 2: Process Msg 2
               Break
        Msg 3: Process Msg 3
               Break
    END CASE
END DO
```

```
Callback ()
{
    Process response
    from Thread B
}
```

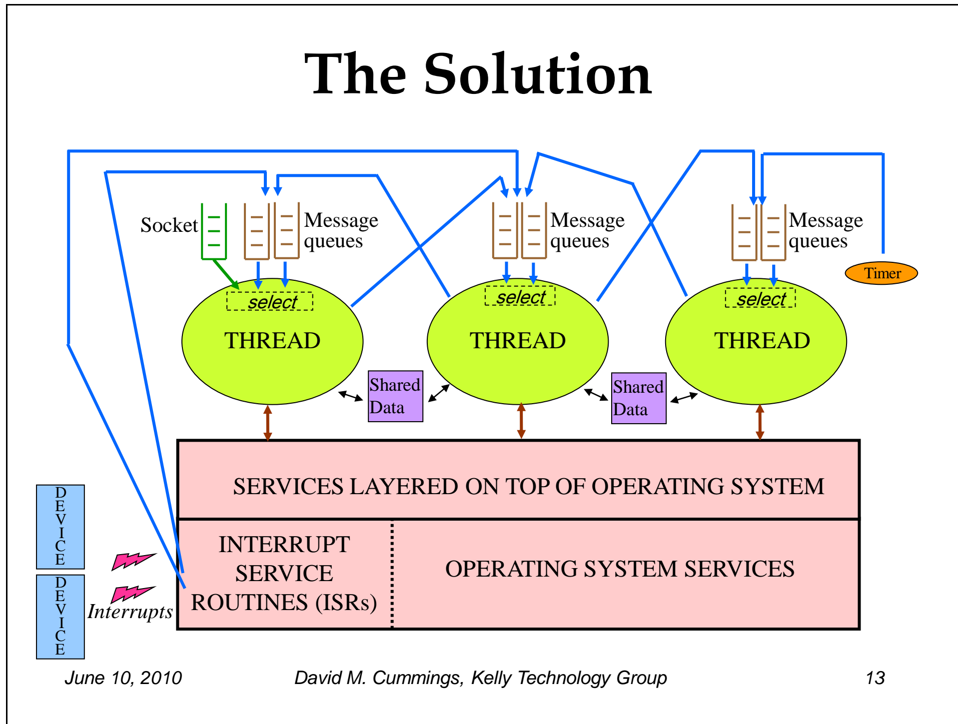
June 10, 2010 *David M. Cummings, Kelly Technology Group* *11*

The Solution

Wait for ALL events (not just messages from other threads) at only one place

```
INITIALIZE
DO Forever
    Wait for all possible events
    CASE event
        Event X: Process Event X
                 Break
        Event Y: Process Event Y
                 Break
        Event Z: Process Event Z
                 Break
    END CASE
END DO
```

June 10, 2010 *David M. Cummings, Kelly Technology Group* *12*

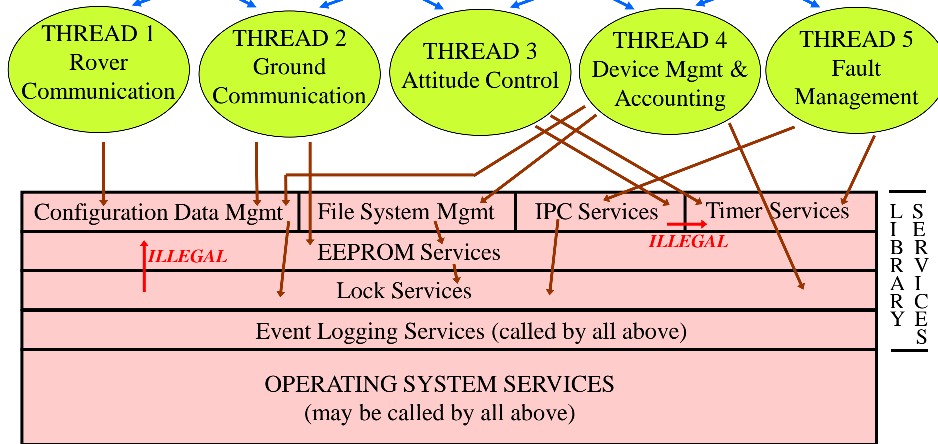

The Solution
Socket
Message queues
Message queues
Message queues
Timer
select
select
select
THREAD
THREAD
THREAD
Shared Data
Shared Data
SERVICES LAYERED ON TOP OF OPERATING SYSTEM
INTERRUPT SERVICE ROUTINES (ISRs)
OPERATING SYSTEM SERVICES
DEVICE
DEVICE
Interrupts
June 10, 2010
David M. Cummings, Kelly Technology Group
13

Layering

- Unlayered systems
  - Any object can call any other object
  - Locks may be held across these calls
  - Unintended recursion
  - Deadlocks

Layering

- Layered systems
  - To avoid unintended recursion and reduce deadlock risk
  - Thread-based objects
    - Can “call” each other using message passing
    - Can call downward
  - Library objects
    - Can only call downward
  - Do not call a method of another object while holding a lock


Example Layering
THREAD 1
Rover
Communication
THREAD 2
Ground
Communication
THREAD 3
Attitude Control
THREAD 4
Device Mgmt &
Accounting
THREAD 5
Fault
Management
Configuration Data Mgmt
File System Mgmt
IPC Services
Timer Services
ILLEGAL
EEPROM Services
ILLEGAL
Lock Services
Event Logging Services (called by all above)
OPERATING SYSTEM SERVICES
(may be called by all above)
LIBRARY SERVICES
June 10, 2010
David M. Cummings, Kelly Technology Group
16

Conclusions – Selected Features for Supporting Thread-Based Implementations

- Create uniform way for thread to wait once for all possible events
- Enforce that a thread waits at only one place
- Enforce that callbacks only deposit messages into message queues
- Layering enforcement: Disallow up-calls and illegal horizontal calls
- Locking enforcement: Disallow calling method of another object while holding a lock

Conclusions – Selected Features for Supporting Thread-Based Implementations

- Asynchronous call (message passing) enforcement
  - Asynchronous method only callable by callers expecting this behavior
- Prevent deadlocks due to full message queues
  - Prevent ISR from ever encountering a queue-full condition
- Explicit support for implementing state machines within threads
- Map these approaches into object-oriented implementations in C++ or Java

BACKUP SLIDES
June 10, 2010
David M. Cummings, Kelly Technology Group
19

Concurrency Control (Locking)

- Not needed for data within thread-based objects
- Needed for shared data within library objects
  - Perform explicit locking
  - Always acquire multiple locks in same order
  - Never call a method of another object while holding a lock

Bugs and Observability

- There will be bugs, no matter how much testing is done
- Employ highly aggressive error detection
  - Test return codes and/or catch exceptions as close as possible to source
  - Liberal assertions, including:
    - Array bounds checks, null pointer checks, data structure integrity checks
    - Leave assertions in production code
  - For bugs, main purpose is fault isolation, not fault recovery
    - Ambitious error recovery logic is likely to be buggy itself
    - Trying to continue in an indeterminate state is dangerous
    - Capture as much state information surrounding an error as possible, including stack trace, and then restart
- Make event logging services easy to use, e.g., make them look like *printf* or *cout*

Bugs and Observability

- Instrument the code to capture as many metrics about the running system as practical, e.g.:
  - High water mark of messages queued to each message queue
  - How many times a writer was blocked because each queue was full, and the max / avg wait time
  - How many times a thread was blocked waiting for each lock, and the max / avg wait time
  - How many times messages of type *x* were sent (to another thread, or to another computer)
  - How many times messages of type *y* were received (from another thread, or from another computer)
  - How many bytes of payload were sent to computer *A*
  - How many bytes of payload were received from computer *A*
  - Etc.

Bugs and Observability

- This can be invaluable during testing and in the field, in order to:
  - determine if the software is behaving as expected
  - see if data structures such as message queues and buffers are sized correctly
- It can form the foundation for some level of automated diagnosis or resource adjustment
  - E.g., warnings when message queues exceed a certain occupancy level, or when a lock is becoming a bottleneck
  - E.g., dynamically increasing the size of message queues

Other Issues and Considerations

- Expose blocking and asynchronous message passing to callers
- Blocking library methods should have timeout options, including zero timeout (non-blocking)
  - Examples are TCP sends (to detect flow control), lock requests (to detect deadlock), message queue writes (to detect deadlock)
- Allocate all dynamic resources (memory, threads) at initialization, to avoid the possibility of resource depletion at runtime
  - Allows runtime resource depletion to be treated as a fatal error (report-and-restart)
- Memory protection – Very desirable for fault containment and fault isolation, but adds design choice: process versus thread
- Hold locks for the minimum amount of time required
  - Where possible, avoid holding a lock when calling any method, even an internal method
  - Never call a blocking method while holding a lock, other than to acquire another lock
- Do as little processing as possible within ISRs

Other Issues and Considerations

- Do as little processing as possible at kernel level (if OS supports user/kernel partitioning)
- Library objects can have threads behind them, but these are subject to library layer rules and should provide synchronous semantics
- Make event logging services easy to use (*printf*-like or *cout*-like) to encourage liberal use
- Select an OS that supports priority inheritance to prevent priority inversion
- Don't misuse thread priorities - they are for meeting deadlines, not for synchronization
- Implement hardware and software watchdogs
- Save reboot reasons and state across reboots
- Prevent reboot loops and process restart loops
- Don't return error status or propagate an exception to a caller if the caller can't do anything about it (e.g., because of a bug in the called method)
- ...

June 10, 2010 *David M. Cummings, Kelly Technology Group* *25*

More Complete List of Features for Supporting Thread-Based Implementations

- Create uniform way for thread to wait once for all possible events
- Enforce that a thread only waits at one place
- Enforce that callbacks only deposit messages into message queues
- Layering enforcement
  - Disallow up-calls and illegal horizontal calls
- Locking enforcement
  - Ensure that sharable data is thread-safe
  - Disallow calling method of another object while holding a lock
  - Enforce lock ordering
  - Disallow holding lock while calling a method unless called method declares OK
  - Disallow holding lock while calling blocking method other than method to acquire a lock
- Blocking enforcement
  - Blocking-intolerant caller can't call method that may block
  - Match caller's delay tolerance with method's upper bound of execution time
- Asynchronous call (message passing) enforcement
  - Asynchronous method only callable by callers expecting this behavior
- Prevent deadlocks due to full message queues
  - Prevent ISR from ever encountering a queue-full condition

June 10, 2010 *David M. Cummings, Kelly Technology Group* *26*

More Complete List of Features for Supporting Thread-Based Implementations

- Implement memory protection and user/kernel partitioning
- Support accumulation and display/dump of metrics
  - Integrate metrics into automated diagnosis to warn of potential problems, e.g., message queues or buffers sized incorrectly, locks becoming a bottleneck
  - Integrate metrics into automated resource reallocation facility, e.g., resizing message queues
- Uniform event logging/tracing
  - callable from interrupt/kernel level and user level
  - stack trace and preservation of information across restarts
- Enforce that all dynamic resources are acquired at initialization
- Explicit support for implementing state machines within threads
- Explicit support for watchdog timers
- Explicit support for preventing process restart loops and reboot loops
- Support priority inheritance for prevention of priority inversion
- Map these approaches into object-oriented implementations in C++ or Java

June 10, 2010 *David M. Cummings, Kelly Technology Group* *27*

Threads Within Library Objects

- A library object can have one or more threads behind it if necessary
  - These threads must obey layering: can't call higher layer, including thread-based objects
- Callers of library objects typically expect synchronous behavior
  - When call returns, operation has completed
  - Maintain these semantics even when one or more threads are hidden behind the library object
- Example is EEPROM writing
  - Thread-based object calls library method *eep_write_data()* to write a block of bytes to EEPROM
    - *eep_write_data()* must wait for ack message from ISR for each byte written
    - *eep_write_data()* then does read-and-compare, and re-writes byte on miscompare
    - Also, only one in-progress write allowed by EEPROM device at any one time
    - Accomplished by introducing another thread behind *eep_write_data()* that manages the writes, and waits for messages from ISR
  - From caller's perspective, *eep_write_data()* is a blocking call that returns when the EEPROM block write has completed

Some Interesting Bugs Involving Concurrency or Networking

- Mars Pathfinder "2+2=5" check
  - Knew a certain integer calculation would always produce an even result, but checked that the result was even anyway
- Intended for detection of radiation-induced hardware errors in CPU
  - Never expected to see it fail, much less during lab testing (with no radiation effects)
  - But we saw it in the lab, only once
- Turned out to be due to a bug in the kernel's context switch software
  - Was not preserving the carry bit correctly across interrupts
- If we did not have the "2+2=5" check in the software (which seemed to some like overkill at the time), we might not have caught this prior to launch
  - Results could have been catastrophic

Some Interesting Bugs Involving Concurrency or Networking

- Mars Pathfinder priority inversion error (well publicized)
- Saw this after spacecraft had successfully landed
  - Symptom was occasional reboots of flight computer due to fatal error declared because software detected a missed deadline
- Able to reproduce at JPL after 18 hours of testing
- Using instrumentation at JPL (which was also on the spacecraft), were able to see that a priority inversion problem was causing the missed deadline
- We had selected this RTOS partly due to its support for priority inheritance, in order to prevent just such errors
- We had used the priority inheritance option on all appropriate semaphores
- Using a tool we wrote, we had verified prior to launch that priority inheritance worked correctly and prevented priority inversion situations that otherwise would have occurred regularly in the flight software
- BUT – the priority inversion that bit us on the surface of Mars was related to a semaphore within the RTOS code itself, which we were not aware of
- Patched the software on the spacecraft to enable the priority inheritance option for this semaphore, and the problem went away

June 10, 2010 *David M. Cummings, Kelly Technology Group* *30*

Some Interesting Bugs Involving Concurrency or Networking

- Saw occasional bit errors in copying large data sets (tens or hundreds of Gbytes) between storage arrays using our backup appliance
- Symptom was one bit flip out of hundreds of billions of bits
- First occurred at customer site, but we were able to reproduce in our lab
- The software involved was byte-oriented, not bit oriented, so we were at a loss to explain it in software
- Were using TCP to do the data transfer, so if bit flips occurred during the transfer we expected TCP to detect it and recover
- Explored a number of theories over the course of many days, but couldn't get to the bottom of it
- Eventually, started looking at the specs for the NIC used in our appliance
- Discovered that it had a TCP Offload Engine (TOE), even though it was a commodity NIC (not for iSCSI)
- This meant that TCP only protected the data up to the NIC, not all the way to main memory
- Disabled the TOE option (vendor provided a way to do this), forcing the TCP layer to execute within the host CPU, and the problem disappeared