Real-Time Software

Exceptions and Low-level Programming

René Rydhof Hansen

2 November 2010

Today's Goals

- Exceptions and Exception Handling (briefly)
- Low-level programming
  - Input/output
  - Language support
  - Memory management

Part II

Exceptions and Exception Handling

General Requirements for Exception Handling

- R1 The facility must be simple to understand and use
- R2 The code for exception handling should not obscure understanding of the program's normal error-free operation
- R3 The mechanism should be designed so that run-time overheads are incurred only when handling an exception
- R4 The mechanism should allow the uniform treatment of exceptions detected both by the environment and by the program
- R5 The exception mechanism should allow recovery actions to be programmed

Exception Handling

- Error return
  - Functions return a value outside "normal" range, indicating success/failure
  - Classic C/UNIX/POSIX
  - Alternative: use global/shared error variable for error indication
- Non-local goto
  - In C: `setjmp` and `longjmp`
  - Requires more "clean-up" than local goto, e.g., call stack
- "High-level" exceptions

Exception Handling in Real-Time Java

- Exceptions in Java are integrated into the OO model
- In Java, all exceptions are subclasses of the predefined class `java.lang.Throwable`
- The language also defines other classes, for example: `Error`, `Exception`, and `RuntimeException`
- Not all blocks can have exception handlers
- Domain exception handlers must be explicitly indicated using `try/catch`

```
try {
        // statements which may raise exceptions
}
catch (ExceptionType e) {
        // handler for e
}
```

- Real-Time Java supports termination model

Resumption versus termination model

- Should the invoker of the exception continue its execution after the exception has been handled
- If the invoker can continue, then it may be possible for the handler to cure the problem that caused the exception to be raised and for the invoker to resume as if nothing has happened
- This is referred to as the resumption or notify model
- The model where control is not returned to the invoker is called termination or escape
- Clearly it is possible to have a model in which the handler can decide whether to resume the operation which caused the exception, or to terminate the operation; this is called the hybrid model

The Resumption Model

- Problem: it is difficult to repair errors raised by the RTS
- Eg, an arithmetic overflow in the middle of a sequence of complex expressions results in registers containing partial evaluations; calling the handler overwrites these registers
- Implementing a strict resumption model is difficult, a compromise is to re-execute the block associated with the exception handler; Eiffel provides such a facility.
- Note that for such a scheme to work, the local variables of the block must not be re-initialised on a retry
- The advantage of the resumption model comes when the exception has been raised asynchronously and, therefore, has little to do with the current process execution

Exception Propagation

- If there is no handler associated with the block or procedure
  - regard it as a programmer error which is reported at compile time
  - but an exception raised in a procedure can only be handled within the context from which the procedure was called
  - eg, an exception raised in a procedure as a result of a failed assertion involving the parameters
- CHILL requires that a procedure specifies which exceptions it may raise (that is, not handle locally); the compiler can then check the calling context for an appropriate handler
- Java allows a function to define which exceptions it can raise; however, unlike CHILL, it does not require a handler to be available in the calling context

Alternative Approach

- Propagating the exception: Look for handlers up the chain of invokers; the Ada and Java approach
- A problem occurs where exceptions have scope; an exception may be propagated outside its scope, thereby making it impossible for a handler to be found
- Most languages provide a catch all exception handler
- An unhandled exception causes a sequential program to be aborted
- If the program contains more than one process and a particular process does not handle an exception it has raised, then usually that process is aborted
- However, it is not clear whether the exception should be propagated to the parent process

Part III

Low-level Programming

Hardware Input/Ouput Mechanisms

- Input/output performed through device registers
- Two approaches to I/O: port mapped and memory mapped

Port mapped I/O

- Requires special I/O bus and I/O address space (ports)
- Special machine instructions
- Better (high granularity) control (potentially)
- More explicit, easier to detect automatically (program analysis)

Memory mapped I/O

- Special addresses in normal memory address space
- Uses same (logical) bus as normal data traffic
- Only uses normal memory access instructions

Controlling Input/Output

Polling (aka. status driven)

- Active polling of device status
- Historically cheaper than interrupt driven
- More deterministic than interrupt driven
- Three kinds of instructions (typically) needed
  1. Test operations
  2. Control operations
  3. I/O operations

Interrupt-driven

- Device coomunicates by requesting an interrupt
- More efficient/better utilisation (typically) than polling
- Several variations
  1. Interrupt-driven program-controlled
  2. Interrupt-driven program-initiated (I/O controllsers, DMA)
  3. Interrupt-driven channel-program controlled (I/O coprocessor)

Controlling I/O: Interrupt-driven I/O

Interrupt-driven program-controlled I/O

- Device requests interrupt when data is ready
- Running task suspended
- I/O performed
- Suspended task resumed
- More non-deterministic than polling, esp. when interrupts are unbounded

Interrupt-driven program-initiated I/O

- Program asks DMA to perform I/O
- Data transfer handled by DMA
- DMA requests interrupt when transfer is complete
- DMA uses memory cycles: cycle stealing
- May add to non-determinism

Controlling I/O: Interrupt-driven I/O

Interrupt-driven channel-program controlled

- Used by I/O co-processor
- Three major components
  1. The hardware "channel"
  2. The channel program
  3. I/O instructions
- May increase non-determinism

Interrupt-driven I/O: Requirements

- Context-switching mechanism
- Interrupt device identification
- Interrupt identification
- Interrupt control
- Priority control

Hardware support for context switch

- Three levels (commonly):
  1. Basic: only program counter (PC)
  2. Partial (most common): PC and program status word (PSW)
  3. Complete: full context (rare)
- Basic and partial often requires extra support in software
- Complete may be too specialised

Interrupt-driven I/O: Requirements

Interrupt device identification

- What device generated an interrupt?
- Especially problem for multiplexed devices
- Interrupt vector (IV) table: each device (or device function) is assigned separate entry in IV table
- Status: necessary information stored in shared memory or accesible through status instructions; useful for generalised ISR
- Polling: when interrupt occurs, ask everyone who did it
- High-level language primitive: automatic translation (often using above techniques at a lower level) to high-level concept, e.g., message or event.

Interrupt-driven I/O: Requirements

Interrupt identification

- Why was an interrupt generated?
- Information communicated through (shared memory) status word or by binding a device to multiple interrupts

Interrupt Control

- Disabling/enabling interrupts form a particular device
- May be controlled individually by interrupt masking
- May be organised in a hierarchy
  - Devices (currently) running at a lower logical level cannot interrupt when the processor (currently) runs at a higher logical level

Priority Control

Prioritising devices, often controlled using the interrupt control facility.

Language Support for Low-level Programming

Modularity and encapsulation

- In particular: ways to encapsulate non-portable parts of the program; typically the low-level parts
- Not only relevant for RTSs!

Abstract models of device handling

- Manipulating device registers
- Representation of interrupts
  - Procedure call
  - Sporadic task invocation
  - Asynchronous notification
  - Shared-variable condition synchronization
  - Message based synchronization

Language Support: Real-Time Java

Manipulating device registers

- Direct access to memory through the `RawMemoryAccess`
- Can only be used for simple types, i.e., not for user defined objects

```
public class ControlAndStatusRegister {
  RawMemoryClass rawMemory;

  public ControlAndStatusRegister(long base, long size)
    rawMemory=RawMemoryAccess.create(IO_Page,base,size);
  }
  public void setControlWord(short value) {
    rawMemory.setShort(0,value);
  }
  ...
  channel = 6;
  shadow = (channel << 8) | start | enable;
  csr.setControlWord(shadow);
```

Memory Management in RTSJ: RawMemoryAccess

```
public class RawMemoryAccess {
  protected RawMemoryAccess(RawMemoryAccess memory,
                            long base, long size);

  public static RawMemoryAccess
      create(java.lang.Object type, long size)
      throws SecurityException, OffsetOutOfBoundsException,
             SizeOutOfBoundsException,
             UnsupportedPhysicalMemoryException;

  public static RawMemoryAccess create(java.lang.Object type,
                                       long base, long size)
      throws SecurityException, OffsetOutOfBoundsException,
             SizeOutOfBoundsException, UnsupportedPhysicalMemoryExc

  public byte getByte(long offset)
     throws SizeOutOfBoundsException, OffsetOutOfBoundsException;

  public void setByte(long offset, byte value)
     throwsSizeOutOfBoundsException, OffsetOutOfBoundsException;
}
```

Language Support: Real-Time Java

Interrupt Handling

- RTSJ views an interrupt as an asynchronous event
- The interrupt is equivalent to a call of the fire method
- The association between the interrupt and the event is achieved via the `bindTo` method in the `AsyncEvent` class
- The parameter is of string type, and this is used in an implementation-dependent manner—one approach might be to pass the address of the interrupt vector
- When the interrupt occurs, the appropriate handler's fire method is called
- Now, it is possible to associate the handler with a schedulable object and give it an appropriate priority and release parameters

Language Support: Real-Time Java

Interrupt Handling

```
AsyncEvent Interrupt = new AsyncEvent();
AsyncEventHandler InterruptHandler = new
      BoundAsyncEventHandler(priParams,
                             releaseParams,
                             null, null, null);

Interrupt.addHandler(InterruptHandler);
Interrupt.bindTo("0177760");
```

Language Support: C/Real-Time POSIX

Low-level Programming in C

By design.

Memory Management

- Memory often very limited in RTSs
- Heap must be managed
  - Manually, e.g., C: `malloc` and `free`
    - Hard to get right
  - Garbage collected, e.g., Java
    - May lead to unpredictable timing behaviour
    - Future: real-time garbage collection
- Stack must be managed
  - Often through worst case memory consumption (WCMC) analysis
  - Similar to WCET analysis

Memory Management in Real-Time Java

1. MemoryArea
    1. HeapMemory (singleton)
    2. ImmortalMemory
    3. ImmortalPhysicalMemory
    4. ScopedMemory
        1. LTMemory
        2. VTMemory

Memory Management in Real-Time Java: MemoryArea

```
public abstract class MemoryArea {
  protected MemoryArea(long sizeInBytes);

  public void enter(java.lang.Runnable logic);
  // associate this memory area to the current thread
  // for the duration of the logic.run method

  public static MemoryArea getMemoryArea(java.lang.Object object);
  // get the memory area associated with the object
  public long memoryConsumed();
  // number of bytes consumed in this memory area
  public long memoryRemaining();
  // number of bytes remaining
  ...
  public synchronized java.lang.Object newInstance(
         java.lang.Class type)throws IllegalAccessException,
            InstantiationException, OutOfMemoryError;
  // allocate an object

  public long size(); // the size of the memory area
}
```

ImmortalMemory

- Immortal memory is shared among all threads in an application
- Objects created in immortal memory are never subject to garbage collection and are freed only when the program terminates

```
public final class ImmortalMemory
  extends MemoryArea
{
  public static ImmortalMemory instance();
}
```

- `ImmortalPhysicalMemory` has the same characteristics as immortal memory but allows objects to be allocated from within a range of physical addresses

ScopedMemory

- A memory area where the objects have a well-defined lifetime
- May be entered explicitly (through the use of the `enter` method) or implicitly by attaching it to a `RealtimeThread` at thread creation time
- Associated with each scoped memory is a reference counter which is incremented for every call to `enter` and at every associated thread creation
- It is decremented when the `enter` method returns and at every associaated thread exit
- When the reference counter reaches zero, all objects residing in the scoped memory have their finalization method executed and the memory is reclaimed
- Scoped memory can be nested by nested calls to `enter`

ScopedMemory

```
public abstract class ScopedMemory
  extends MemoryArea
{
  public ScopedMemory(long size);

  public void enter(java.lang.Runnable logic);

  public int getMaximumSize();

  public MemoryArea getOuterScope();

}
```

ScopedMemory

To avoid the possibility of dangling pointers, a set of access restrictions are placed on the use of the various memory areas

Heap objects can reference other heap objects and objects in immortal memory only (i.e. it cannot access scoped memory)

Immortal objects can reference heap objects and immortal memory objects only;

Scoped objects can reference heaped objects, immortal objects and objects in the same scope or an outer scope only

ScopedMemory: Example

```
import javax.realtime.*;
public class ThreadCode implements Runnable
{
  private void computation()
  {
    final int min = 1*1024;
    final int max = 1*1024;
    final LTMemory myMem = new LTMemory(min, max);

    myMem.enter(new Runnable()
    {
      public void run()
      {
        // code here which requires access
        // to temporary memory
      }
    } );
  }
```

ScopeMemory: Example (continued)

The thread can now be created; note, no parameters other than the memory area and the `Runnable` are given

```
ThreadCode code = new ThreadCode();

RealtimeThread myThread = new RealtimeThread(
        null, null, null, ImmortalMemory.instance(),
        null, code);
```

Stack Management

- Embedded programmers also have to be concerned with stack size
- Specifying the stack size of a task/thread requires trivial support (for example, in Ada it is via the Storage_Size attribute applied to a task; in POSIX it is via pthread attributes)
- Calculating the stack size is more difficult; as tasks enter blocks and execute procedures their stacks grow
- To estimate the maximum extent of this growth requires knowledge of the execution behaviour of each task
- This knowledge is similar to that required to undertake WCET analysis
- WCET and worst-case stack usage bounds can be obtained from control flow analysis of the task's code

Summary

- Exceptions and Exception Handling (briefly)
- Low-level programming
  - Input/output
  - Language support
  - Memory management