Memory Management for Safety-Critical Java

Martin Schoeberl
Technical University of Denmark

Thanks to SCJ EG and Kelvin Nilsen and Anders Ravn

Safety-Critical Java

- A Java profile for safety-critical applications
- Restricts the expressiveness of RTSJ
- Simpler task model
- Restricted scope model

SCJ Levels

- Three levels for different application areas
    - L0: the ignored cyclic executive ;-)
    - L1: Ravenscar style
    - L2: More dynamics with nested missions

Mission Concept

- A mission consists of
  - A collection of handlers
  - A shared memory (mission memory)
- Missions can be restarted
- Missions can form a sequence
- No real-time constraints on mission start/stop

Memory

- Java depends heavily on dynamic memory allocation
- In normal Java we have garbage collection
  - Convenient tool (more in a later talk)
- RTSJ did not believe in RT garbage collection
  - Scoped memory model

RTSJ Scopes

- Memory area similar to stack allocation
  - Explicit context enter and leave
  - Can be shared between threads
- Issues
  - Live time and pointer assignments
  - Sharing between threads

Scopes in SCJ

- Based in the RTSJ model
  - Managed by the SCJ runtime
  - No explicit creation
  - Extends and restricts the RTSJ classes
- Mission memory and private memory
- Plus we have immortal memory

RTSJ Scope Issue

- Backing store ‘allocation’ (the memory) for scopes is not very well defined
  - A C ‘malloc’ is mentioned in the RTSJ spec.
- Undisciplined usage of scopes leads to memory fragmentation
- Nesting of scopes does not mean nesting of backing store

SCJ Scopes

- Avoid fragmentation
  - Maximum size of backing store needs to be specified
- Restricted scope sharing
  - Mission memory is shared
  - Handler scopes are thread private

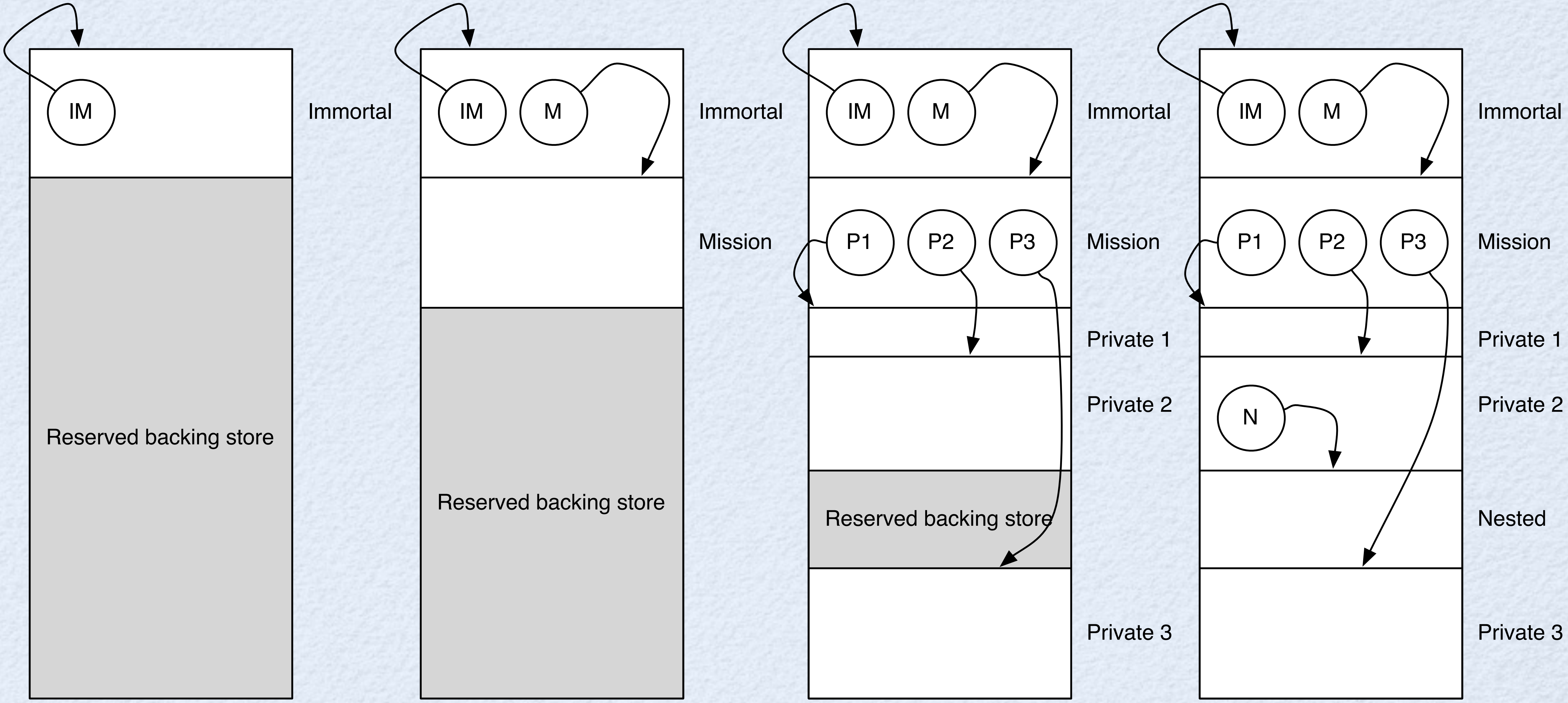
Backing Store Nesting

- The SCJ definition allows a nesting implementation
- Immortal, mission, and private memory give a strict hierarchy
- Nesting in the implementation, not in the contract
- Notion of backing store plus reserved memory

Unified Memory Areas

- Immortal, mission, and private have much in common
- Can be implemented by a single class
- Each inner memory (backing store) is contained in the outer memory

Nested Memory



(1) Safelet startup

(2) Mission creation

(3) Handler creation

(4) Nested private of P2

Implementation

- A single Memory class
  - Not at the SCJ API visible
  - Used by SCJ memory classes
- Represents all memory types
- A nested memory object is allocated in the outer memory

Immortal

- Special as there is no outer context
- An immortal memory area is allocated in immortal - chicken/egg problem
  - Some JVM magic at boot

Implementation

- Track of
  - Start address
  - Local allocations
  - Nested allocation
  - Parent
  - Inner nested scope

```
public class Memory {

        int startPtr;
        int allocPtr;
        int endLocalPtr;
        int allocBsPtr;
        int endBsPtr;
        Memory parent;
        int level;

        Memory inner;

        static Memory immortal;

        Memory() {}

        static Memory getImmortal(int start, int end)

        Memory(int size, int bsSize) {...}
        Memory(int size) {...}

        void enter(Runnable logic) {...}
        void executeInArea(Runnable logic) {...}

        void enterPrivateMemory(int size, Runnable logic)
}
```

Implementation

- On the Java Optimized Processor JVM
- Concept not JOP specific
- Part of system classes
  - Where we can use integers for memory addresses
- Memory area is part of thread context

Conclusions

- SCJ scopes avoid fragmentation
- A scope implementation can use nested allocation
- All memory areas can be represented by a single class
- Implementation is an important step towards SCJ on JOP