Concurrency-mønstre i indlejrede systemer

Temadag om Embeddede systemer
Teknologisk Institut, 6. december 2004

Finn Overgaard Hansen,
Ingeniørhøjskolen i Århus
E-mail: foh@iha.dk

Version: 6-12-2004

Agenda

- Problemet
- Ønskerne
- Løsningsforslag
  - RTOS OO abstraktioner
- Opsummering

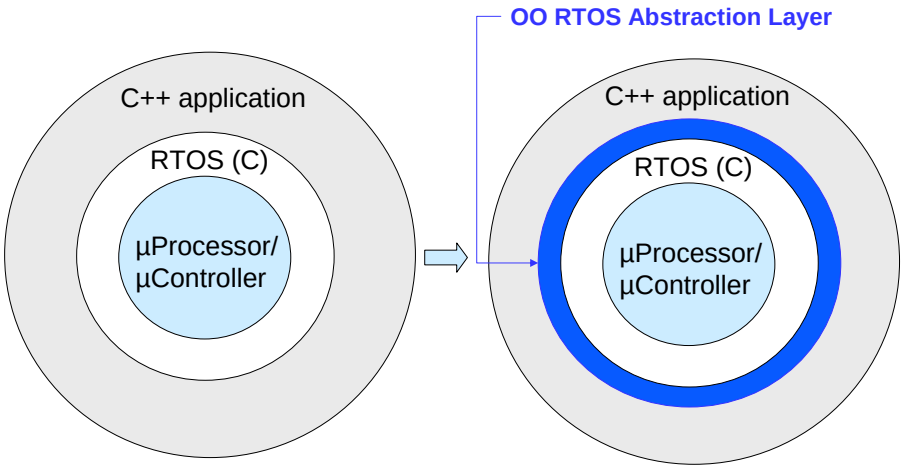
Problemet

- Udvikling af embeddede systemer
  - Programmeres i C og i stigende omfang i C++
  - Stigende anvendelse af Objektorienterede metoder
  - Anvendelse af UML som modelleringssprog og som dokumentation [UML-2.0]
  - Anvendelse af et Realtidsoperativsystem (RTOS), der har et C API
  - Sproget C++ mangler dsv. sproglige konstruktioner, der understøtter multiprogrammering
- Impedansproblemet
  - OOA/OOD modellering af tråde og kommunikation med klasser i C++
  - Implementering vha. RTOSs C-API

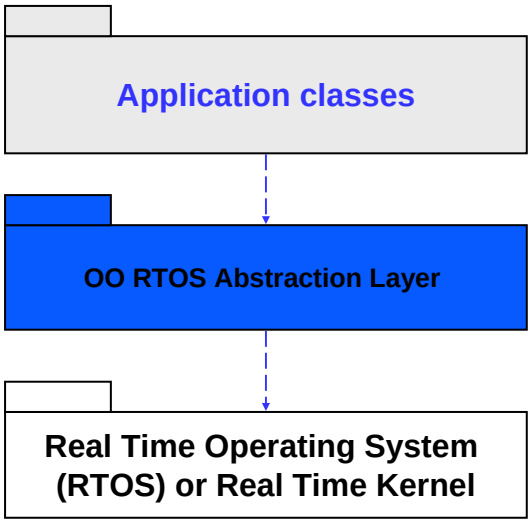
Ønskerne

- Vi vil gerne kunne foretage en sømløs implementering af vores OO design og dets concurrency mekanismer
- Vi vil gerne kunne udvikle embeddede systemer på en udviklingsmaskine (f.eks. en Windows PC eller en Linux PC), der er forskellig fra target-maskinen (µC/µP)
  - Der er ofte langt bedre værktøjer til rådighed på denne platform
- Vi vil gerne kunne teste så meget som muligt af det embeddede system på udviklingsmaskinen
  - Med teststubbe der simulerer den rigtige HW
  - Med anvendelse af maskinens operativsystem for test af concurrency
- Det skal efterfølgende være nemt at flytte applikationen over på targetmaskinen
- Det er ofte også ønskeligt at kunne portere applikationen til en anden targetplatform
  - Medfører meget ofte også skift af RTOS
  - Man ønsker at denne portering kun vedrører få og lokaliserede dele af koden

OO RTOS indkapsling



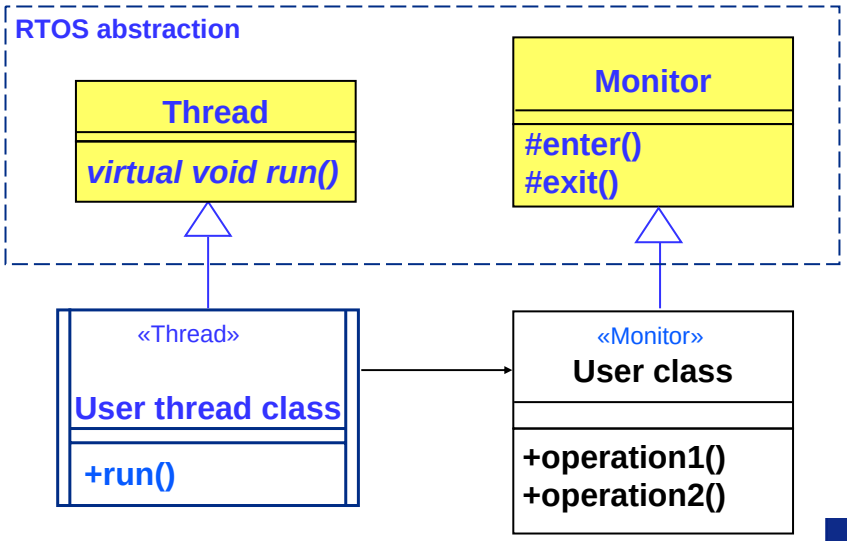
OO RTOS Abstraction Layer



Nogle løsningsmuligheder

- Anvendelse af Concurrency mønstre
  - Thread, Monitor [POSA2], Active Object [POSA2]
  - Udviklet som Wrapper Facades til RTOS, [POSA2] mønster
- Anvendelse af ACE framework [ACE]
  - En C++ open source implementering af [POSA2] wrapper facade mm.
- Anvendelse af et Real-Time CASE værktøj med kodegenerering
  - Rhapsody fra firmaet [ILogix]
  - Rational Rose Technial Developer fra [IBM-Rational] (den tidligere Rational Rose Real-Time)
- JThreads/C++, *"Java-like Threads for C++"*
  - "JThreads/C++" is a registered trademark of IONA Technologies, [IONA], Version 2.0.0b1, 30 Oct 2003.
  - Part of Orbacus ORB – available for Windows, Unix, Linux, Solaris

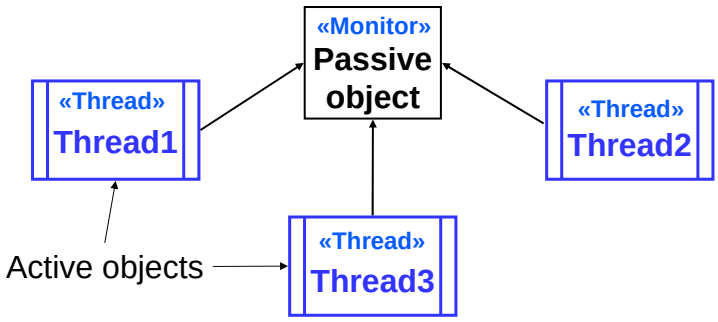
Two useful OS Abstractions



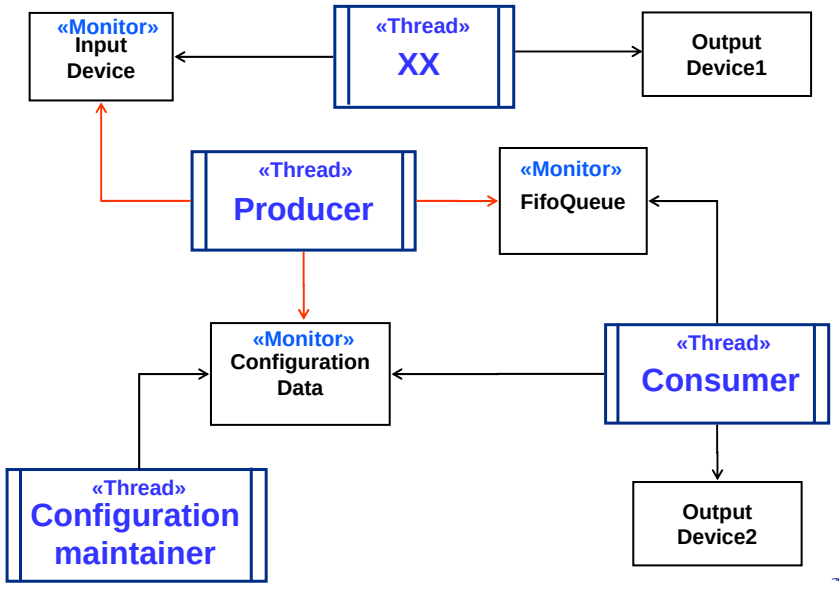
Monitor Object

- Multiple threads of control accessing the same object concurrently

UML Stereotype Annotations



Multiprogrammerings eksempel (UML 2.0)



RTKernel Thread Class (C++)

```
class Thread {
public:
        Thread(char* threadName,int threadPriority,
                int threadStackSize,);
        virtual void run()=0;                   // NB! Pure virtual
        virtual void start();
        void sleep(int milliSecs)
        virtual ~Thread();
        static void RTKAPI threadMapper(void* p) {
            ((Thread *)p)->run();  // call of task loop function }
private:
        int stackSize;
        int priority:
        char* name;
        RTKTaskHandle handle;
};
```

RTKernel: RTOS fra [OnTime]

RTKernel Thread implementation

```
                                        // Inheritance version
Thread::Thread(char* threadName,int threadPriority, int threadStackSize) {
   name= threadName;
   priority= threadPriority;
   stackSize= threadStackSize;
   char* name;
   handle= 0;
}

void Thread::start() {                          // create & start thread
   if (handle ==0)
        handle= RTKRTLCreateThread(Thread::threadMapper,
          priority,stackSize,0, this, name);
}

void Thread::sleep(int milliSec) {
   RTKDelay(CLKMilliSecsToTicks(milliSec));
}
```

Anvendelse af Thread Class i Producer (1)

```
#include “Thead.h”

class Producer : public Thread
{
public:
   Producer(char* threadName,int threadPriority, int stackSize)
    : Thread(threadName,threadPriority,threadStackSize) {}
   ~Producer();
   virtual void run();            // implementation of thread loop
private:
  //
};
```


Anvendelse af Thread Class i Producer (2)

```
class Producer : public Thread
{
public:
   Producer(char* threadName, int threadPriority, int stackSize,
     InputDevice *pID,ConfigurationData *pCD,FifoQueue *pFQ)
      : Thread(threadName,threadPriority,threadStackSize),
        pInputDevice(pID), pConfigurationData(pCD),
        pFifoQueue(pFQ) {}
   ~Producer();
   virtual void run();            // implementation of thread loop
private:
   InputDevice *pInputDevice;
   ConfigurationData *pConfigurationData;
   FifoQueue *pFifoQueue;
};
```

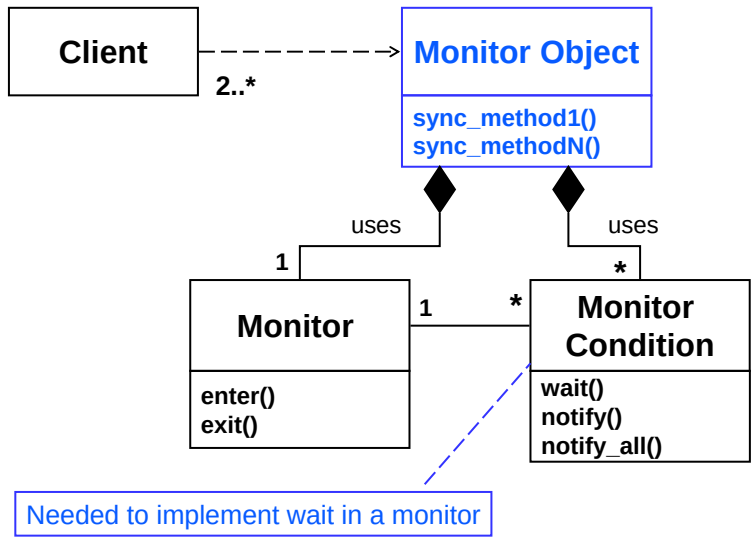

Producer Thread - loop

```
void Producer::run()
{
        int sleepTime;
        while (1)
        {
                sleepTime= pConfigurationData->getSleepTime();
                speed= pInputDevice->getValue();
                // do some calculation
                pFifoQueue->put(speed);
                sleep(sleepTime);
        }
}
```


Example of a main program

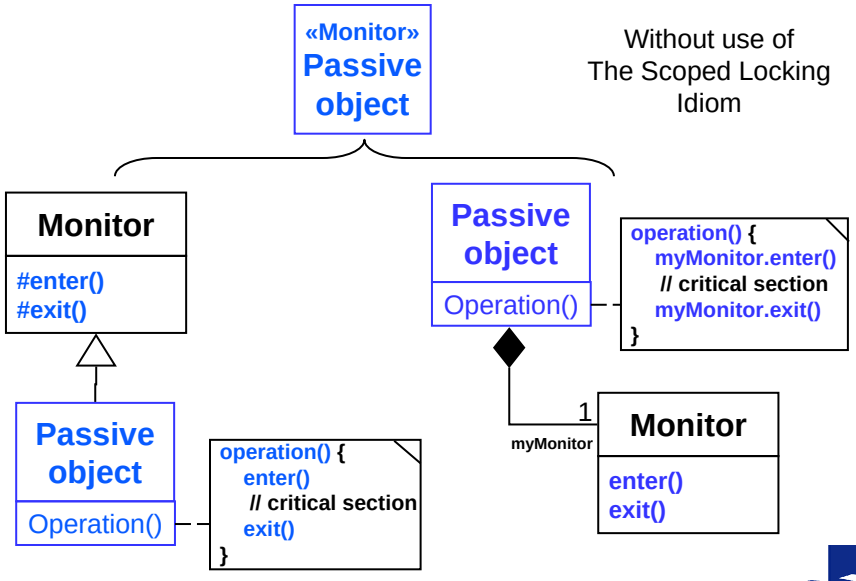
```
int main()
{
   // create monitor objects
   InputDevice inputDevice;
   ConfigurationData configurationData;
   FifoQueue fifoQueue;

   // create Producer thread
   Producer myProducer("ProducerThread", MAX_PRIORITY,
         StackSize5k, &inputDevice, &configurationData,&fifoQueue);
   // create other threads
         // Consumer myConsumer = ….
   // start treads
   myProducer.start();
   myConsumer.start();
   // main task loop    while (1) ;
};
```


Monitor Object Pattern Structure (POSA2)



[POSA2]

Implementation Strategier for simpel Monitor



Problemer med Monitor operationer

- Every public monitor operation accessed by clients should start with acquiring the lock (**enter** monitor or **acquire**) and end with releasing the lock (**exit** monitor or **release**)
- The programmer could forget it in some return path
- The solution is not exception-safe

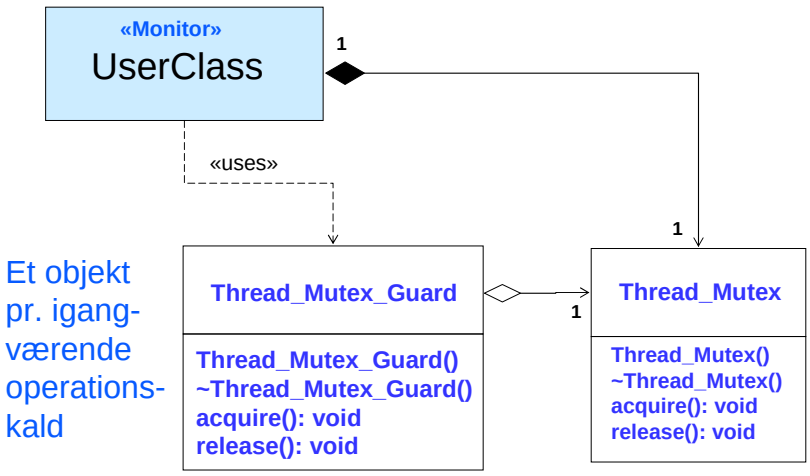
Solution: Scoped Locking C++ Idiom (POSA2)

- Define a **guard class** whose **constructor** acquires a lock when control enters a scope and whose **destructor** automatically releases the lock when control leaves the scope

```
void MonitorClassX::operation()
{
   Thread_Mutex_Guard myGuard(*threadMutex);
    // …  Critical section code
    // …
    return;
}   // destructor called automatically
```

[POSA2]

Anvendelse af Guard class



Et objekt pr. igang-værende operations-kald

acquire/release eller enter/exit

[POSA2]

Thread_Mutex_Guard Class

```
class Thread_Mutex_Guard {
public:
   Thread_Mutex_Guard(Thread_Mutex &lock)
                          : lock_(&lock), owner_(false) { acquire(); }
   void acquire() {lock_->acquire(); owner_=true; }
   void release() {
      if (owner_) {
         owner_=false; lock_->release();  }
   }
   ~Thread_Mutex_Guard() { release(); }
private:
   Thread_Mutex *lock_;  // Pointer to our lock
   bool owner_;          // is lock held by this object ?

    // disallowing copy and assignment
   Thread_Mutex_Guard(const Thread_Mutex_Guard &);
   void operator=(const Thread_Mutex_Guard &);
};
```

[POSA2]

Thread_Mutex Example (Solaris imp.)

```
class Thread_Mutex {
public:
   Thread_Mutex()    { mutex_init(&mutex_,USYNC_THREAD,0); }
    ~Thread_Mutex() { mutex_destroy(&mutex_); }
   void acquire()       { mutex_lock(&mutex_); }
   void release()       { mutex_unlock(&mutex_); }
private:
   // Solaris-specific Mutex mechanism
   mutex_t mutex_;

   // disallow copying and assignment
   Thread_Mutex(const Thread_Mutex &);
   void operator=(const Thread_Mutex &);
   friend class Thread_Condition;
};
```

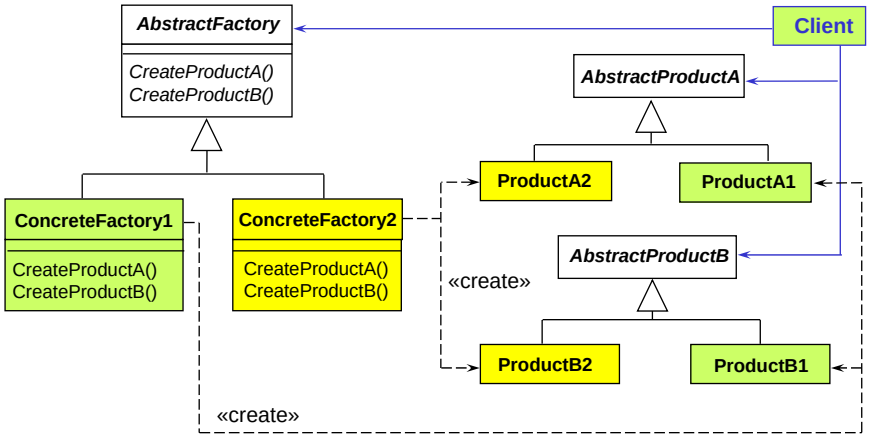
[POSA2]

Tread_Condition Example (Solaris imp.)

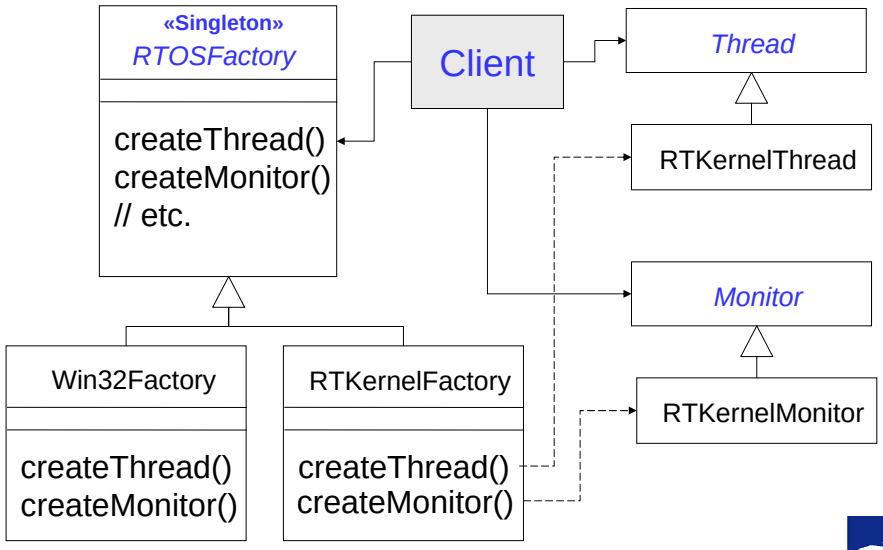
```
class Thread_Condition {
public:
   Thread_Condition(const Thread_Mutex &m) : mutex_(m)
         { cond_init(&cond_, USYNC_THREAD, 0); }
   ~Thread_Condition() { cond_destroy(&cond_); }
   void wait(Time_Value *timeout= 0) {
      cond_timedwait(&cond_, &mutex_.mutex_,
                     timeout==0 ? 0 : timeout->msec());
   }
   void notify() { cond_signal(&cond_); }
   void notify_all() { cond_broadcast(&cond_); }
private:
   condt_t cond_;   // SOLARIS condition variable
   const Thread_Mutex &mutex_;
};
```

[POSA2]

Abstract Factory (GoF)



Anvendelse af AbstractFactory



Main program med abstract factory

```
int main(int argc, char *argv[]) {
   RTOSFactory *rtosFactory;
   // ..
   if (strcmp(osString,"RTKernel")==0)
         rtosFactory= RTKernelFactory::Instance();
   else if (strcmp(osString,"Win32") ==0)
         rtosFactory= Win32Factory::Instance();

   // Create of Monitor objects
   InputDevice inputDevice;
   ConfigurationData configurationData;
   FifoQueue fifoQueue;

   Producer myProducer(&inputDevice,&configurationData,&fifoQueue);
   Thread *producerTread=
         rtosFactory->createThread(myProducer,"ProducerThread",
               HIGH_PRIORITY, StackSize5k);
   // etc.
}
```


To alternative løsninger

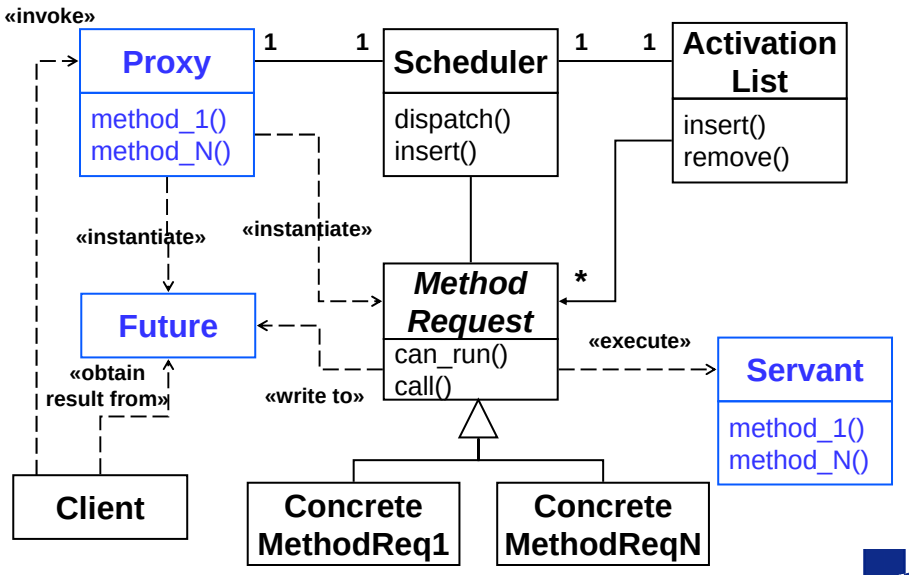
Som i det forrige eksempel:

```
Producer myProducer(&inputDevice,&configurationData,&fifoQueue);

Thread *producerTread=
         rtosFactory->createThread(myProducer,"ProducerThread",
               HIGH_PRIORITY, StackSize5k);
```

Alternativt – integreret i Producer klassen

```
Producer::Producer(&inputDevice,&configurationData,&fifoQueue)
{
   producerTread=
         rtosFactory->createThread(this,"ProducerThread",
               HIGH_PRIORITY, StackSize5k);
}
```


POSA2 Active Object Design Pattern

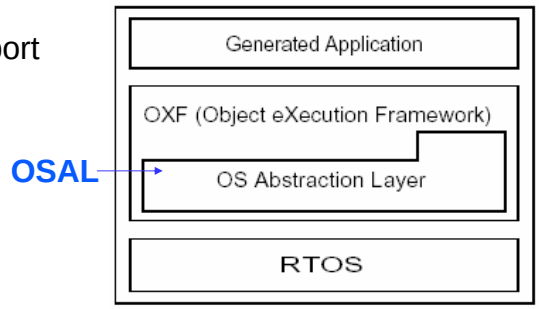


[POSA2]

Rhapsody OSAL: OS Abstraction Layer

The OSAL consists of a set of interfaces (abstract classes) **that provide all the required operating system services** for the application, including:

- Tasking services
- Synchronization services
- Message queues
- Communication port
- Timer service



Rhapsody supporterede operativsystemer

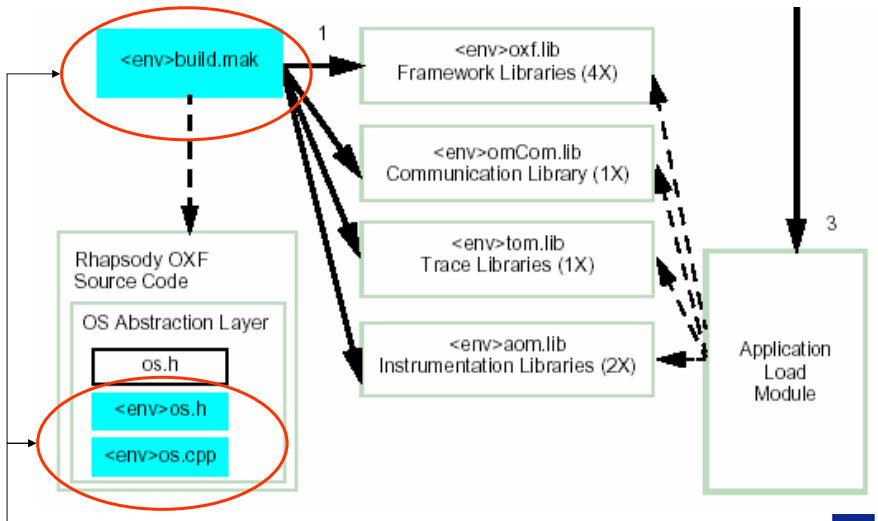
C++
- Borland
- **Linux**
- **Microsoft**, MicrosoftDLL
- **MicrosoftWinCE**
- MSStandardLibrary
- OsePPCDiab, OseSFK
- PsosPPC, **PsosX86**
- QNXNeutrinoCW, QNXNeutrinoGCC
- Solaris2, Solaris2GNU
- **VxWorks**

C
- Microsoft
- PsosPPC
- PsosX86
- VxWorks

Java
- JDK

Portering af Rhapsody til "MIT_OS"

<env>=MIT_OS



Kræver udvikling af disse filer

JThreads/C++ (1)

Java

```
public class HelloWorld extends
   Thread
{
  public void run()
  {
    System.out.println("Hello");
  }

  static public void main(String
   args[])
  {
    Thread t = new HelloWorld();
    t.start();
  }
}
```

C++

```
class HelloWorld : public
   JTCThread
{
public:
  virtual void run()
  {
    cout << "Hello" << endl;
  }
};

int main(int argc, char** argv)
{
  JTCInitialize initialize;
  Thread* t = new HelloWorld;
  t -> start();
  return 0;
}
```

"Java-like Threads for C++"

JThreads/C++ (2)

Java

```
synchronized void writeBuffer()
{
  while(len_ < 80)
  {
    try
    {
      wait();
    }
    catch(InterruptedException ex)
    {
    }
  }
  System.out.write(data_,0,len_);
  System.out.flush();
  len_ = 0;
}
```

C++

```
void writeBuffer()
{
  JTCSynchronized sync(*this);
  while(len_ < 80)
  {
    try
    {
      wait();
    }
    catch(JTCInterruptedException)
    {
    }
  }
  cout.write(data_, len_);
  cout.flush();
  len_ = 0;
}
```


Java Character Buffer (1)

```
public class CharacterBuffer {
      private char[] data_ = null;
      private int len_ = 0;

synchronized public void addChar(char c)
{
      if(data_ == null || len_ == data_.length)
      {
        byte[] newData = new byte[len_+128];
        if (data_ != null)
          System.arraycopy(data_, 0, newData, 0, len_);
        data_ = newData;
      }
      data_[len_++] = c;
}
```

C++ JTreads CharacterBuffer (2)

```
class CharacterBuffer : public JTCMonitor {
      char* data_;
      int len_;
      int max_;
public:
      CharacterBuffer(): data_(0), len_(0), max_(0){}
      ~CharacterBuffer(){delete[] data_; }

void addChar(char c) {
      JTCSynchronized synchronized(*this);
      if (len_ >= max_)
      {
        char* newData = new char[len_+128];
        memcpy(newData, data_, len_);
        delete[] data_;
        data_ = newData;
        max_ += 128;
      }
      data_[len_++] = c;
}
```

Opsummering

- Anvendelse af et OO RTOS abstraktionslag kan medføre en ”sømløs” implementering af et OO design
- Anvendelse af Thread og Monitor konceptet giver en mere sikker (typecheck) og simplere multiprogrammering
- Der findes flere løsningsmuligheder
- Concurrency mønstre er en hjælp ved udvikling af eget OO RTOS lag og ved anvendelse af kommercielle eller Open Source løsninger
- Applikationerne bliver dog afhængige af det anvendte OO RTOS lag

References

- [POSA2] Pattern-Oriented Software Architecture
  - Volume 2. Patterns for Concurrent and Networked Objects. Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Bushmann, Wiley 2000
- [GoF] Design Patterns, Gamma et.al, Addison-Wesley 1995
- [UML-2.0]: Unified Modeling Language Ver. 2.0 – se www.omg.org/uml
- [ACE]: Adaptive Communication Environment - www.cs.wustl.edu/~schmidt/ACE.html
- [IONA]: www.iona.com/devcenter/orbacus/jtc.htm
- [ILogix]: www.ilogix.com
- [IBM-Rational]: http://www-306.ibm.com/software/awdtools/developer/technical/
- [OnTime] Embedded RTOS for x86 systems, http://www.on-time.de/