Thread Abstraction Layer — TAL
In Memoriam of PURE (∗1995 †2002)

Wolfgang Schröder-Preikschat

Friedrich-Alexander University Erlangen-Nuremberg
Department of Computer Science 4
(Distributed Systems and Operating Systems)

www4.cs.fau.de
The PURE de Luxe

- Operating-system engineering in the small: carried to an extreme
  - Playing with capabilities of C/C++ programming systems
- "Pluckiness for sacrilege": an operating system is no end in itself
  - A thread: that's no concern of an operating system — almost...
- To apply Occam’s razor:
  - "Entia non sunt multiplicanda praeter necessitatem."
  - "Entities should not be multiplied beyond necessity."
  - "All other things being equal, the simplest solution is the best."
- Smooth transition: programming-language ↔ operating-system level
PURE de Luxe

- operating-system engineering in the small: carried to an extreme
  - playing with capabilities of C/C++ programming systems
- “pluckiness for sacrilege”: an operating system is no end in itself
  - a thread: that’s no concern of an operating system — almost...
- to apply Occam’s razor:
  - “Entia non sunt multiplicanda praeter necessitatem.”
  - “Entities should not be multiplied beyond necessity.”
    - “All other things being equal, the simplest solution is the best.”
- smooth transition: programming-language ↔ operating-system level
PURE de Luxe

- operating-system engineering in the small: carried to an extreme
  - playing with capabilities of C/C++ programming systems
- “pluckiness for sacrilege”: an operating system is no end in itself
  - a thread: that’s no concern of an operating system — almost...
- to apply Occam’s razor:
  - “Entia non sunt multiplicanda praeter necessitatem.”
  - “Entities should not be multiplied beyond necessity.”
  - “All other things being equal, the simplest solution is the best.”
- smooth transition: programming-language ↔ operating-system level
Specialization without Abandonment of Reusability
Cornerstones of the PURE/CiAO Development Process

**PURE**

family-based design eases extension and contraction of software
  - stepwise *functional enrichment* of system abstractions

feature-based conditioning ensures an application-aware finishing
  - mapping of features to entities of the software generation process
Specialization without Abandonment of Reusability
Cornerstones of the PURE/CiAO Development Process

PURE

family-based design eases extension and contraction of software
  ▶ stepwise functional enrichment of system abstractions

feature-based conditioning ensures an application-aware finishing
  ▶ mapping of features to entities of the software generation process

CiAO

feature modeling distinguishes common from variable system properties
  ▶ identifying commonalities, differences, constraints, and conflicts

aspect-oriented programming improves separation of concerns
  ▶ factorization of cross-cutting concerns by aspect classes
### Specialization without Abandonment of Reusability (cont.)

**Principles of PURE/CiAO Operating-System Engineering**

<table>
<thead>
<tr>
<th>minimal subset of system functions</th>
<th>⇐⇒</th>
<th>superclass</th>
<th>asset</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional enrichment</td>
<td>↓</td>
<td>inheritance</td>
<td>conditioning</td>
</tr>
<tr>
<td>minimal system extensions</td>
<td>⇐⇒</td>
<td>subclass</td>
<td>specialization</td>
</tr>
</tbody>
</table>

**Program family**

<table>
<thead>
<tr>
<th>OOP</th>
<th>AOP</th>
</tr>
</thead>
</table>

**wosch**

**Thread Abstraction Layer — TAL**
Family-Based Design
Minimal System Extensions to a Minimal Subset of System Functions
specialized nuclei an ensemble of different operating modes
- configuration depends on user-required system properties
  - in a functional and non-functional sense
- application programs get what they want—no more and no less

---

**Operating-System Product Line**

Scalable Solution in Terms of Memory Footprint and Execution Time (IA-32)

<table>
<thead>
<tr>
<th>nucleus instance</th>
<th>size (in bytes)</th>
<th>latency (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>text</td>
<td>data</td>
</tr>
<tr>
<td>exclusive</td>
<td>434</td>
<td>0</td>
</tr>
<tr>
<td>interruptive</td>
<td>812</td>
<td>64</td>
</tr>
<tr>
<td>cooperative</td>
<td>1620</td>
<td>0</td>
</tr>
<tr>
<td>non-preemptive</td>
<td>1671</td>
<td>0</td>
</tr>
<tr>
<td>coordinative</td>
<td>1882</td>
<td>8</td>
</tr>
<tr>
<td>preemptive</td>
<td>3642</td>
<td>8</td>
</tr>
</tbody>
</table>
Functional Hierarchy of Thread Abstractions
Layered According to Weight Classes

lightweight thread
↓
featherweight thread
↓
bantamweight thread
↓
flyweight thread
↓
strawweight thread
Minimal Subset of System Functions
Strawweight Thread — Use Case

```c
#include "luxe/Act.h"

#define STACKSIZE 64
#define LEEWAY 16

int main (int argc, char *argv[]) {
    static Act *son, *dad; // thread pointers

    char pool[STACKSIZE]; // spawner provides spawnee stack
    son = new(pool, STACKSIZE - LEEWAY) Act; // spawner makes spawnee instance

    if ((dad = son->assume())) { // spawner clones, spawnee erupts
        for (;;) { // spawnee shares state
            something(); // spawnee does its job
            dad = dad->resume(); // spawnee yields spawner
        }
    }

    son = son->resume(); // spawner yields spawnee
    anything(); // spawner does some other job
}
```
Minimal Subset of System Functions (cont.)
Strawweight Thread — Abstract Data Type

```cpp
#include "luxe/type/size_t.h"
#include "luxe/machine/pc_t.h"

class Act {
protected:
    pc_t tbc; // where to be continued upon resume
public:
    void* operator new (size_t, char*, size_t);
        // return aligned (stack) pointer as "this"
    Act* assume (); // create thread: returns twice (0 spawner, else spawnee)
    Act* resume (); // switch thread
};
```

fundamental threading functions

- thread creation and activation “in passing”
- sharing of the entire processor state, except stack pointer
- supports control-flow switches of different thread weight classes
Minimal Subset of System Functions (cont.)
Strawweight Thread — Implementation for IA-32

```c
#include "luxe/Act.h"

Act* Act::assume () {
    asm ("movl 4(%esp), %eax"); // read stack pointer of spawnee
    asm ("movl (%esp), %edx");  // grab return address of spawner
    asm ("movl %edx, (%eax)" ); // pass as start address to spawnee
    return 0;                  // indicate return from spawner
}

Act* Act::resume () {
    register Act* aux;
    asm ("movl %esp, %0 : ":r" (aux)); // remember stack pointer
    asm ("movl 4(%esp), %esp");    // switch stack
    return aux;                   // resume thread, return forerunner
}
```

idea (use `-fomit-frame-pointer`, never inline)
- hand return address down to some inactive flow of control
- provide a function whose sole task is to exchange the stack pointer
Minimal System Extensions
Thread Weight Classes and Abstraction of Different Weightily Threads

```cpp
#include "luxe/machine/ActMode.h"

enum FluxVariety {
    Strawweight = Act,
    Flyweight = GPR|OVR|OFP, // save all except volatile and FPU registers
    Bantamweight = GPR|OVR, // save all except volatile registers
    Featherweight = GPR|BMR, // save all using block move, if applicable
    Lightweight = GPR // save all
};
```

```cpp
#include "luxe/Act.h"
#include "luxe/machine/FluxVariety.h"

template<FluxVariety T>
class Flux : public Act {
public:
    Act* induce (Flux<T>*&); // create thread and inherit processor state
    Act* unwind (Act&); // switch thread, performed inline
    Act* resume (Act&); // switch thread: maps to unwind()
};
```
#include "luxe/Flux.h"
#include "luxe/machine/ActState.h"

template<FluxVariety T>
inline Act* Flux<T>::induce (Flux<T>*& scion) {
    Act* clade; // spawner thread pointer
    if ((clade = assume())) // spawner clones, spawnee erupts
        return resume(*clade); // spawnee adopts state, yields spawner
    scion = (Flux<T>*)Act::resume(); // spawner yields spawnee
    return 0; // spawner indicates its return
}

idea

- give new flow of control the chance to inherit some processor state
- save ones processor state before giving control back to creator
#include "luxe/Flux.h"
#include "luxe/machine/ActState.h"

template<FluxVariety T>
inline Act* Flux<T>::unwind (Act& next) {
    Act* peer;
    if (T & SOS) {  // save processor state onto runtime stack...
        ActState<T|BMR> *apr;  // pointer to saved processor state
        apr = ActState<T|BMR>::stack();  // push processor state onto stack
        peer = next.resume();  // switch thread
        apr->clear();  // pop processor state from stack
    } else {  // save processor state into buffer variable...
        ActState<T|BMR> apr;  // save buffer for processor state
        apr.cache();  // write processor state into buffer
        peer = next.resume();  // switch thread
        apr.apply();  // read processor state from buffer
    }
    return peer;  // return forerunner
}
#include "luxe/Flux.h"

#define STACKSIZE 256
#define LEEWAY 16

typedef Flux<Bantamweight> Fibre;

int main (int argc, char *argv[]) {
    char pool[STACKSIZE]; // spawnee stack space

    Fibre *son = new(pool, STACKSIZE - LEEWAY) Fibre; // spawnee thread
    Act   *dad; // spawner thread

    if ((dad = son->induce(son))) { // spawner clones
        for (;;) {
            something(); // spawnee does its job
            dad = son->resume(*dad); // spawnee yields
        }
    }

    son = (Fibre*)((Fibre*)dad)->unwind(*son); // spawner yields
    anything(); // spawner does its job
}
### Family of Thread Abstractions

Memory Footprints of Fundamental Thread Switching Functions (IA-32)

<table>
<thead>
<tr>
<th>weight class</th>
<th>static</th>
<th>dynamic</th>
<th>subtotal</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>straw</td>
<td>8 + 7</td>
<td>4 + 4</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>fly</td>
<td>11 + 11</td>
<td>8 + 12</td>
<td>42</td>
<td>65</td>
</tr>
<tr>
<td>bantam</td>
<td>11 + 13</td>
<td>8 + 16</td>
<td>48</td>
<td>71</td>
</tr>
<tr>
<td>feather</td>
<td>11 + 7</td>
<td>8 + 32</td>
<td>58</td>
<td>81</td>
</tr>
<tr>
<td>light</td>
<td>11 + 19</td>
<td>8 + 28</td>
<td>66</td>
<td>89</td>
</tr>
</tbody>
</table>

listed are...

- static (text, no data in this case) and dynamic (stack) requirements
- needs for function call (left term) and function body (right term)
From now on it’s all plain sailing... ;-

Functional Hierarchy of an Operating-System Family

<table>
<thead>
<tr>
<th>layer</th>
<th>function</th>
<th>concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>program management</td>
<td>text, data, overlay</td>
</tr>
<tr>
<td>9</td>
<td>mass-storage management</td>
<td>partition, file, file system</td>
</tr>
<tr>
<td>8</td>
<td>process management</td>
<td>activity, context, stack</td>
</tr>
<tr>
<td>7</td>
<td>memory management</td>
<td>segment, page</td>
</tr>
<tr>
<td>6</td>
<td>information interchange</td>
<td>packet, message, channel, portal</td>
</tr>
<tr>
<td>5</td>
<td>device control</td>
<td>signal, character, block, stream</td>
</tr>
<tr>
<td>4</td>
<td>access protection</td>
<td>segment, page, domain, capability</td>
</tr>
<tr>
<td>3</td>
<td>resource sharing</td>
<td>lock, semaphore, monitor</td>
</tr>
<tr>
<td>2</td>
<td>job/task scheduling</td>
<td>energy, event, priority, time slice</td>
</tr>
<tr>
<td>1</td>
<td>control-flow exchange</td>
<td>coroutine, interrupt, continuation</td>
</tr>
</tbody>
</table>

wosch
Thread Abstraction Layer — TAL
A penny saved is a penny got...  
Operating Systems for Embedded Systems

\{\text{BlueCat, HardHat}\} \text{ Linux, Embedix, Windows } \{\text{CE, NT Embedded}\}, \ldots

- not \{\text{adaptable, customizable, scalable, small, sparingly}\} enough
A penny saved is a penny got...

Operating Systems for Embedded Systems

\{BlueCat, HardHat\} Linux, Embedix, Windows \{CE, NT Embedded\}, \ldots

- not \{adaptable, customizable, scalable, small, sparingly\} enough

\ldots, BOSS, C\{51, 166, 251\}, CMX RTOS, Contiki, C-Smart/Raven, eCos, eRTOS, Embos, Ercos, Euros Plus, Hi Ross, Hynet-OS, ITRON, LynxOS, MicroX/OS-II, Nucleus, OS-9, OSE, OSEK \{Flex, Plus, Turbo, time\}, Precise/\{MQX, RTCS\}, proOSEK, pSOS, PURE, PXROS, QNX, Realos, RTMOSxx, Real Time Architect, RTA, RTOS-UH, RTXC, Softune, SOS, SSXS RTOS, ThreadX, TinyOS, VRTX, VxWorks, \ldots

- over 50 % of OS for the embedded-systems market are proprietary
Summary

PURE

- highly reusable and yet specialized operating-system assets must not be a contradiction in terms
- key to success: (embedded) operating system as a program family
Summary

PURE

- highly reusable and yet specialized operating-system assets must not be a contradiction in terms
- key to success: (embedded) operating system as a program family

CiAO extends on PURE by an aspect-aware design

- focus is on increasing configurability by means of AOP
  - especially wrt. architectural and non-functional properties
- application of AOP principles from the very beginning
  - kernel developed in AspectC++
Conclusion
Final Remarks
Resume

Thread Abstraction Layer — TAL