O
O S E
E

TAL — Object Orientation

Operating-System Engineering

Thread Abstraction Layer — TAL

• implementation of \{fly, feather, light\}weight threads by three components:
  – thread yard ................................................................. 2
  – thread executive ........................................................... 9
  – thread interface extension .............................................. 29

• supplementing components take care of platform-specific features:
  – central processing unit .................................................... 16
    * register access functions ........................................... 19
    * state buffering ............................................................ 24
  – application binary interface .......................................... 14

• development of the class hierarchy and code fragments exemplary to C++
Thread Yard — thy

- at the physical level, a thread is represented by an untyped stack pointer
  - for being able to execute, at minimum a thread needs a stack as resource
  - when suspended, the thread's stack stores the thread's resumption address

- at the logical level, a thread is represented by a typed (aligned) stack pointer
  - an inactive thread's stack pointer refers to a well-defined object
  - this object captures the resumption address of the thread
  - it is automatically created when a thread suspends execution

- a class ensemble takes care of mapping the logical to the physical view

Thread Yard

thyThread

- abstraction from a thread's resumption address (ra)
  - the ra attribute is a text-segment pointer (PC): it is valid only when the thread is inactive
  - the this member variable serves as stack pointer: it gets defined when a thread becomes inactive

- a thyThread instance (thus, ra) is by-product of suspending thread execution
  - static member functions\(^1\) take care of ra definition and {con,de}struction

- a thyThread instance disappears implicitly when thread execution is resumed

\(^1\)In the following, static member functions are indicated by a trailing caret (i.e., ').
Thread Yard

- support thread instantiation by a single method:
  - labeling of the current thread’s stack
  - splitting up the current thread of control
  - returning a thread handle (0 → dad, 0 → son)

- a minimal extension to thyThread aiming at improved “user friendliness”
  - compared to thyThread, this may be at the expense of performance:
    * spawn() exploits a C++ ?:clause to generate the return value
    * in turn, exploitation of spawn() is, typically, part of an if-clause
  - as a consequence, a nested conditional expression needs to be evaluated
Thread Yard

- generation of a typed stack pointer:
  1. CPU-defined alignment
     - `top()'`
  2. CPU-defined stack expansion
     - `top()'`
  3. user-defined type conformance
     - `new()` resp. `new[]()`

- attempts of stack-pointer deallocation by exploitation of `delete()` are trapped

Thread Yard

- description of a thread's runtime stack
  - a user-defined number of `thyEntity` slots
- represented as a parameterized data type
  - with the number of slots as actual parameter
- manifests a `thyThread` including stack space
Thread Executive — the

- thread execution happens cooperatively and proceeds in a coroutine-like manner.
- thread execution is minimal extension of a lightweight thread by saving its resumption address when switching to another thread.
- lightweight thread minimal extension of a lightweight thread by saving its processor context when switching between threads.
- lightweight thread minimal extension of a lightweight thread by saving its processor context when switching between threads.
- extensions of lightweight threads into featherweight and lightweight threads.
- lightweight thread minimal extension of a lightweight thread by saving its processor context when switching between threads.
Thread Executive

- implementation of a featherweight thread
  - attribute is the thread’s context pointer
    * identifies the “scope” of the thread
  - thread shifts update the context pointer

- thread construction means binding of a thread instance to a thread context
  - the context pointer (toc²) makes up the thread’s initial stack pointer

- thread shifting updates the binding by pointing to the saved resumption address

²thread of control

Thread Executive

- implementation of a lightweight thread
  - depends on stack space for context saving

- threads save/restore their context on their own
  - before/after the thread-shifting procedure

- context saving distinguishes between the abstract and concrete processor
  - grant() is for the abstract processor “compiler”
  - yield() is for the concrete processor “CPU”

Operating-System Engineering — T.A.E. — Object Orientation
Thread Executive

- the initial thread resp. “objective main()”
  - distinguishes different invocation patterns:
    * with no parameters
    * with an *argument vector*
    * with an *environment pointer*
  - provides the default entry point(s)
- introduced to support pure object-orientation
- the “genius” is automatically created by main(), with the action() left open

Thread Executive

**theGenius**

+action()
+action(int,char**)
+action(int,char**,char**)

Class Hierarchy

```
theThread
  -thyEntity* toc
  +theThread()
  +theThread(thyEntity*)
  +scope() : thyEntity*
  +scope(thyEntity*)
  +shift (thyThread*)

theObject
  +theObject()
  +theObject(thyEntity*)
  +grant (thyThread*)
  +yield(thyThread*)

abiStack

cpuStack

theGenius
  +action()
  +action(int,char**)
  +action(int,char**,char**)```
Application Binary Interface — abi

- the interface between application and hardware as defined by the compiler
  - particularly, it assigns to selected CPU registers a specific function
    * e.g., to employ the x86 general-purpose register “ebp” as local base
  - it specifies which compiler feature is implemented by which CPU feature

- a major aspect that has influence on a thread model is register banking:
  
  **volatile register set** i.e. the set of those CPU registers whose contents need not be maintained across procedure calls
  
  **non-volatile register set** i.e. the set of those CPU registers whose contents must be maintained across procedure calls

- thread-context saving may take advantage out of this “artificial” distinction

---

Application Binary Interface

- context save and restore
  - stack-based
  - **non-volatile register set**

- platform dependencies:
  - GNU gcc compilers
  - Intel x86 processors

- minimization of context-saving overhead in the light of big (RISC) register sets
Central Processing Unit — cpu

- the processor-dependent interface to the underlying physical machine
  - defines the CPU’s register set and specific CPU control functions
  - defines register access functions (RAF) for language-level manipulations
- an instance of the respective CPU abstraction(s) exists by default
  - namely the “hardware object” that represents the actual CPU
  - creation and {con,de}struction of that object at runtime is meaningless
- some kind of “support layer” to control CPU operation by C++ programs

Central Processing Unit — cpuStack

- context save and restore
  - stack-based
  - general-purpose registers
  - dedicated instructions

```
inline void cpuStack::store () {
  asm("pushal");
}
inline void cpuStack::clear () {
  asm("popal");
}
```

- platform dependency: Intel x86 processor family (in the present case)
  - compiler dependency exists only w.r.t. the inline assembler feature
- the overhead for context save/restore may be fairly high with RISC processors
  - e.g., a PPC: 32 (32-bit) integer and 32 (64-bit) floating-pointer registers
Central Processing Unit

cpuModel (1)

- major aspect is to access a CPU register by a register access function (RAF)
  - each register is considered a class having two public operators:

```
struct RAF {
    type operator = (const type);  // write into CPU register
    operator type () const;       // read from CPU register
}
```

- whereby `type` specifies the width (in bits) of the particular register

- there are as many RAF definitions as CPU registers that need to be accessible\(^3\)

\(^3\)RAF definition and implementation is a typical case for automation. Otherwise, manual work will end up in "imposition". Macro-based programming mitigates the efforts somewhat. For example, the struct shown is a "#define _CLASS(name,type)" with "name" equals RAF (rafClass.h).

Register Access Functions

```
class cpuMode1GPR {
    __CLASS(EDI, bits32);
    __CLASS(ESI, bits32);
    __CLASS(EBP, bits32);
    __CLASS(ESP, bits32);
    __CLASS(EBX, bits32);
    __CLASS(EDX, bits32);
    __CLASS(ECX, bits32);
    __CLASS(EAX, bits32);
    cpuMode1GPR ();
    cpuMode1GPR (const cpuMode1GPR&);
    public:
    EDI edi;
    ESI esi;
    EBP ebp;
    ESP esp;
    EBX ebx;
    EDX edx;
    ECX ecx;
    EAX eax;
};
```

```
class cpuMode1SR {
    __CLASS(CS, bits16);
    __CLASS(DS, bits16);
    __CLASS(SS, bits16);
    __CLASS(ES, bits16);
    __CLASS(EIP, bits16);
    cpuMode1SR ();
    cpuMode1SR (const cpuMode1SR&);
    public:
    CS cs;
    DS ds;
    SS ss;
    ES es;
};
```

```
class cpuMode1PR {
    __CLASS(Flags, bits16);
    cpuMode1PR ();
    cpuMode1PR (const cpuMode1PR&);
    public:
    Flags flags;
};
```
Register Access Functions
machine-x86/inline/cpuModel\{GP,S,F\}R.h

```c
#include "SEIZEbits32.h"

inline bit16 cpuModelPR::FlagS::operator = (const bit16 aux) {  
    asm volatile ("push %0" : : "g" (aux));  
    return aux;  
}

inline cpuModelPR::FlagS::operator bit16 () const {  
    bit16 aux;  
    asm ("pushf");  
    asm volatile ("pop %0" : "=g" (aux));  
    return aux;  
}

inline bit16 cpuModelPR::FlagS::operator = (const bit16 aux) {  
    asm volatile ("movl %0, %1" : : "g" (aux));  
    return aux;  
}

inline cpuModelPR::FlagS::operator bit16 () const {  
    bit16 aux;  
    asm ("popf");  
    return aux;  
}
```

Register Access Functions
machine-x86/inline/rafSeize.h

```c
#define --SEIZEbits32(type, name, what)  
inline bits32 name::operator bits32 () const {  
    return aux;  
}
```

## Operating-System Engineering — T.A.L — Object Orientation
# Register Access Functions

```c
#include "Cpu.h"

extern int foo();

int main () {
  bits16 flags = cpu->flags;
  bits16 ds = cpu->ds;
  Cpu->ds = Cpu->cs;
  *(int*)foo = 0x3C3;
  Cpu->flags = flags;
  Cpu->eax = 4711;
  return foo();
}
```

## Exploitation

```c
#include "machine/cpuModel.h"
#define Cpu ((cpuModel*)42)
```

## CPU Register Access Functions

- `cpuModelGPR`
  - `cpuModelGPR()`
  - `cpuModelGPR(const cpuModelGPR&)`
    + EDI edi
    + ESI esi
    + EBP ebp
    + ESP esp
    + EBX ebx
    + EDX edx
    + ECX ecx
    + EAX eax

- `cpuModelFR`
  - `cpuModelFR()`
  - `cpuModelFR(const cpuModelFR&)`
    + Flags flags

- `cpuModelSR`
  - `cpuModelSR()`
  - `cpuModelSR(const cpuModelSR&)`
    + CS cs
    + DS ds
    + SS ss
    + ES es

## CPU Model (2)

- `inline void cpuModel::block () {` ...
- `inline void cpuModel::admit () {` ...

---

Operating-System Engineering — TAL — Object Orientation
State Buffering

- buffered save/restore of a thread’s ABI state
  - RAF is used for CPU-register access
  - read/write register contents
  - the state buffer is not stack-bound

- ABI-state manipulation then can be accomplished in the following way:
  1. flush the ABI state into the buffer by reading the CPU registers
  2. access selected ABI registers within the buffer
  3. apply the buffer contents to the CPU by writing to the CPU registers

```cpp
inline void abiImage::flush () {
    (*this)[ABI_EDI] = cpu->edi;
    (*this)[ABI_ESI] = cpu->esi;
    (*this)[ABI_EBP] = cpu->ebp;
    (*this)[ABI_EBX] = cpu->ebx;
}

inline void abiImage::apply () const {
    Cpu->edi = (*this)[ABI_EDI];
    Cpu->esi = (*this)[ABI_ESI];
    Cpu->ebp = (*this)[ABI_EBP];
    Cpu->ebx = (*this)[ABI_EBX];
}

inline bits32& abiImage::operator [] (int slot) const {
    return (bits32&)gpr[slot];
}
```
State Buffering

- buffered save/restore of a thread’s CPU state
  - the *state buffer* is not stack-bound

- CPU-state manipulations imply the following:
  1. flush the CPU state into the buffer
  2. access selected registers within the buffer
  3. apply the buffer contents to the CPU

- RAF abstractions are used to read/overwrite the contents of the CPU registers

```cpp
inline void cpuImage::flush () const {
    (*this)[CPU_EAX] = cpu->eax;
    (*this)[CPU_ECX] = cpu->ecx;
    (*this)[CPU_EDX] = cpu->edx;
    (*this)[CPU_ESI] = cpu->esi;
    (*this)[CPU_ESP] = cpu->esp;
    (*this)[CPU_EDI] = cpu->edi;
}
```

```cpp
inline void cpuImage::apply () const {
    cpu->edi = (*this)[CPU_EDI];
    cpu->esi = (*this)[CPU_ESI];
    cpu->ebp = (*this)[CPU_EBP];
    cpu->esp = (*this)[CPU_ESP];
    cpu->ebx = (*this)[CPU_EBX];
    cpu->edx = (*this)[CPU_EDX];
    cpu->ecx = (*this)[CPU_ECX];
    cpu->eax = (*this)[CPU_EAX];
}
```

```cpp
inline bits32 cpuImage::operator [] (int slot) const {
    #ifdef hackOperatorSubscription
        return ((bits32*)this)[slot];
    #else
        return (slot < ABI_NGPR) ? (*abiImage*)this[slot] : ((bits32*)gpr[slot - ABI_NGPR]);
    #endif
}
```
State Buffering

```c
void fooBar () {
    CPULmage foo;
    foo.flush();
    ...
}
```

```c
CPULmage bar;
void fooBar () {
    ...
    foo[CPU_EAX] = foo[CPU_EBP] - foo[CPU_ESP];
    bar = foo;
    ...
}
```

Thread Interface Extension — tie

- logical separation of the thread abstraction(s) from the user abstraction(s)
  — encapsulation of thread instantiation and activation by a single method
- different ways of interfacing are provided by different (minimal) extensions

To make the user thread a

```
\{ pointer to virtual default \}
\{ parametrized member \}
```

- the abstractions provided are optional and aim at improving “user friendliness”
Thread Interface Extension

- **pointer to parametrized function**
  - parameter is the function's thread
    * i.e., a `tieThing*`
  - an "objectless" function otherwise

- ordinary C functions become threads:

  ```c
  void gadget (tieThing* self) {
    self->grant(...);
  }
  ```

- member functions of derived (single-inheritance path) classes may work also

  > This is a "hack", absolutely dirty, not recommendable—yet efficient, but it depends on the C++ compiler.

---

Thread Interface Extension

- **pointer to member function**
  - parameter is the function's thread
    * i.e., a `theObject&`
  - any method of a user-defined class

- C++ methods become threads:

  ```c
  void T::entry (theObject& self) {
    self.grant(this->next);
  }
  ```

- kind of **delegation**, with member-function execution by its own private thread
Thread Interface Extension
tieActor

- **virtual function** (i.e., late binding)
  - redefined methods become threads
    
    ```cpp
    void Foo::action () {
        grant(this->next);
    }
    ```

- a user thread is considered the final specialization of the system's abstraction(s)
  - it extends the system thread by user-defined attributes and methods

- the price to be paid is the presence of (at least) two virtual-function tables

Thread Interface Extension
tieBeing

- **default function**
  - member of a user-defined class
    
    ```cpp
    void T::action () {
        this->self->grant(next);
    }
    ```

- the (minimal) system extension supports the *inlining* of the member function
  - depending on the function's specification/implementation—and the compiler

- clients benefit from all (public) features of the system and user abstraction(s)