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:

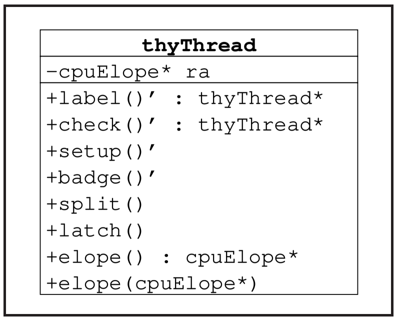
- supplementing components take care of platform-specific features:

- 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., ’).

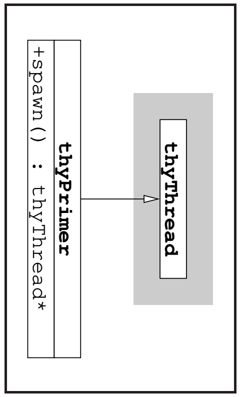
machine-x86/inline/thyThread.h

```
#include "cpu.h"

inline void thyThread::setup () {
  asm ("pushl $1f");
}

inline void thyThread::badge () {
  asm ("1:");
}

inline thyThread* thyThread::check () {
  setup();
  return (thyThread*)(bits32)cpu->sp;
}

inline thyThread* thyThread::label () {
  thyThread* toc;
  asm volatile ("leal -4(%%esp),%0" : "=g" (toc));
  return toc;
}

inline void thyThread::split () {
  setup();
  cpu->sp = (bits32)this;
  badge();
}

inline void thyThread::latch () {
  cpu->sp = (bits32)this;
  asm ("ret");
}
```


Thread Yard

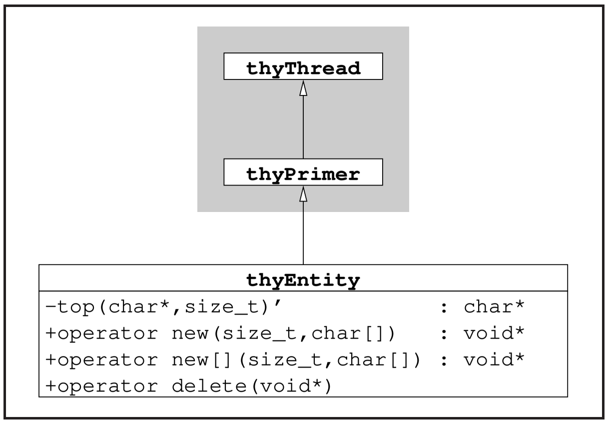
thyPrimer

- 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)



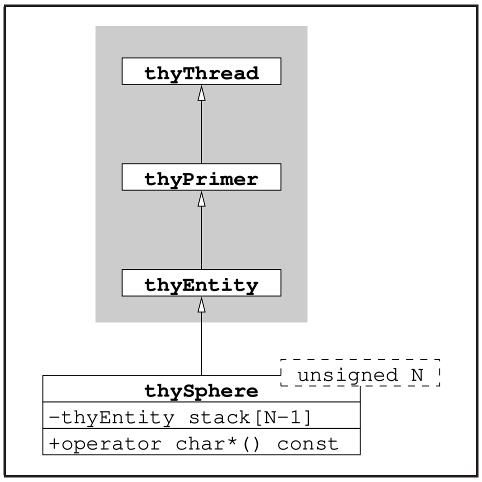
- a minimal extension to `thyThread` aiming at improved "user friendliness"
  - compared to `thyThread`, this may be at the expense of performance:
    * `spawn()` exploits a C++ ?-clause to generate the return value
    * in turn, exploitation of `spawn()` is, typically, part of an `if`-clause
  - as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

Thread Yard `thyEntity`



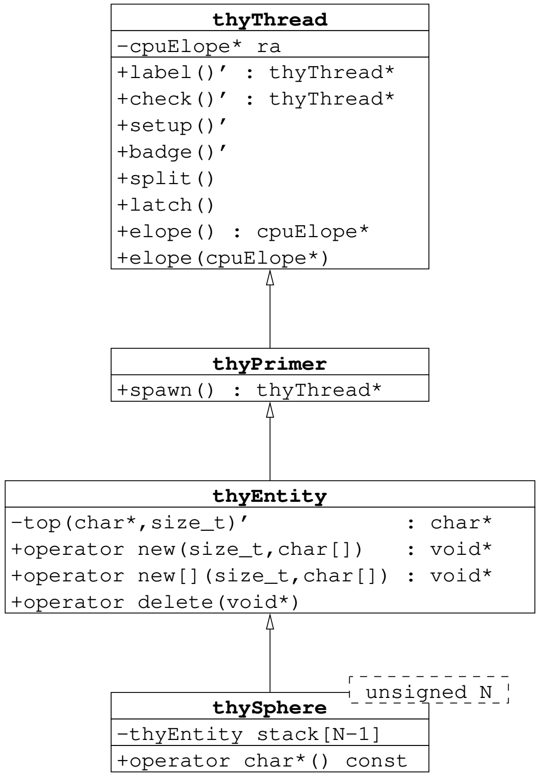
- generation of a *typed stack pointer*:
  1. CPU-defined alignment
     - `top()'`
  2. CPU-defined stack expansion
     - `top()'`
  3. user-defined type conformance
     - `new()` resp. `new[]()`
- attempts of stack-pointer deallocation by exploitation of `delete()` are trapped

Thread Yard `thySphere`



- 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

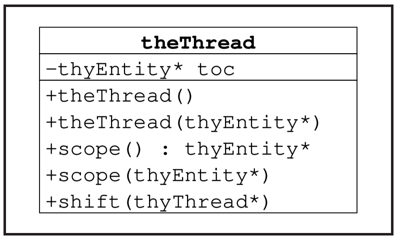
Thread Yard — Class Hierarchy



Thread Executive — `the`

- extensions of flyweight threads into featherweight and lightweight threads:
  - **flyweight thread** the minimal basis (`thy`), saves its resumption address when switching to another thread
  - **featherweight thread** minimal extension of a flyweight thread by saving its context pointer before switching to another thread
  - **lightweight thread** minimal extension of a featherweight thread by saving/restoring its processor context when switching between threads
- thread execution happens cooperatively and proceeds in a coroutine-like manner

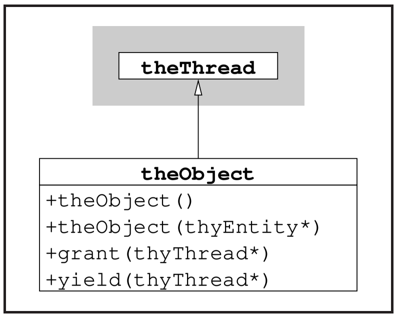
Thread Executive `theThread`



- 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 (`toc`[2]) makes up the thread's initial stack pointer
- thread shifting updates the binding by pointing to the saved resumption address

[2] thread of control

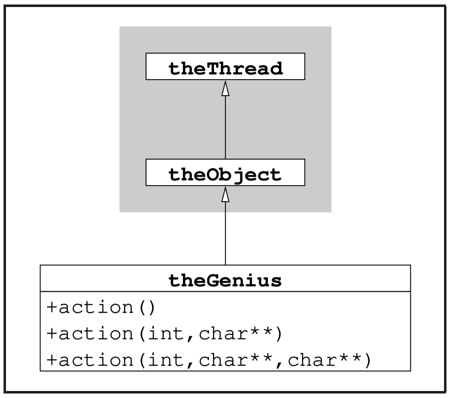
Thread Executive `theObject`



- 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" $\neg$*inline*
  - `yield()` is for the concrete processor "CPU" *inline*

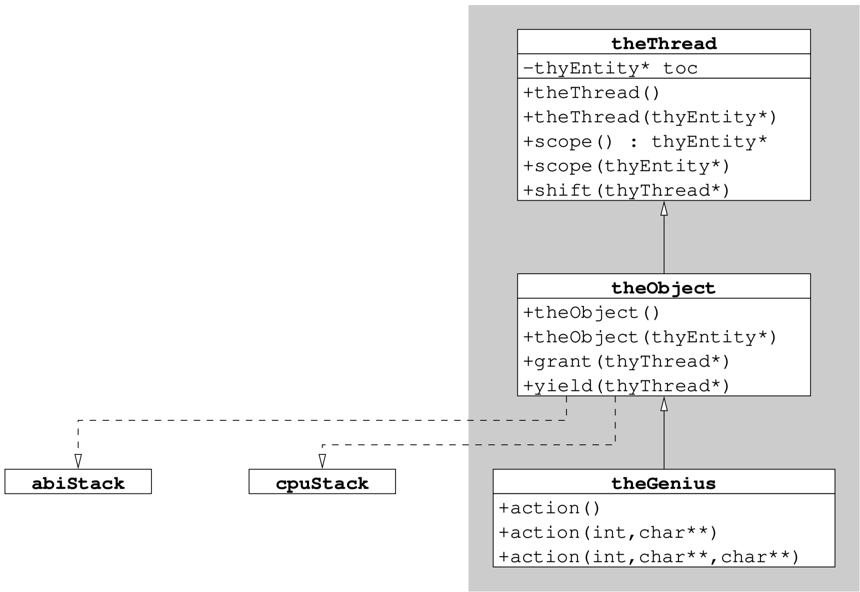
Thread Executive — `theGenius`

- 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 — Class Hierarchy



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*:

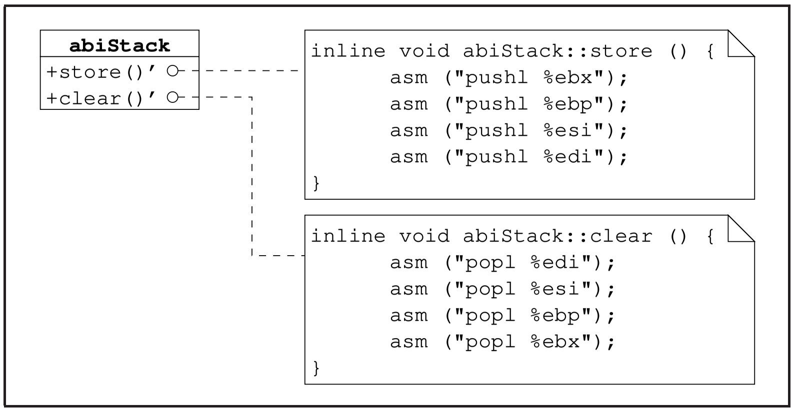
  volatile register set i.e. the set of those CPU registers whose contents need not be maintained across procedure calls

  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 `abiStack`

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



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

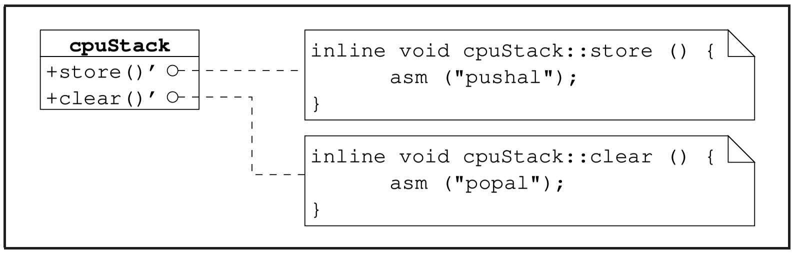
- minimization of context-saving overhead in the light of big (RISC) register sets

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



- 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 — x86

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

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

```
class cpuModelFR {
  __CLASS(Flags,bits16);
  cpuModelFR ();
  cpuModelFR (const cpuModelFR&);
public:
  Flags flags;
};
```

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

```
#define __SEIZEbits8(type,name,what)\
inline bits8 type::name::operator = (const bits8 aux) {\
  asm volatile ("movb %0,%%" #what "" : : "g" (aux));\
  return aux;\
}\
inline type::name::operator bits8 () const {\
  bits8 aux;\
  asm volatile ("movb %%" #what ",%0" : "=g" (aux));\
  return aux;\
}

#define __SEIZEbits16(type,name,what)\
inline bits16 type::name::operator = (const bits16 aux) {\
  asm volatile ("mov %0,%%" #what "" : : "g" (aux));\
  return aux;\
}\
inline type::name::operator bits16 () const {\
  bits16 aux;\
  asm volatile ("mov %%" #what ",%0" : "=g" (aux));\
  return aux;\
}

#define __SEIZEbits32(type,name,what)\
inline bits32 type::name::operator = (const bits32 aux) {\
  asm volatile ("movl %0,%%" #what "" : : "g" (aux));\
  return aux;\
}\
inline type::name::operator bits32 () const {\
  bits32 aux;\
  asm volatile ("movl %%" #what ",%0" : "=g" (aux));\
  return aux;\
}
```


Register Access Functions machine-x86/inline/cpuModel{GP,S,F}R.h

```
__SEIZEbits32(cpuModelGPR,EDI,edi);
__SEIZEbits32(cpuModelGPR,ESI,esi);
__SEIZEbits32(cpuModelGPR,EBP,ebp);
__SEIZEbits32(cpuModelGPR,ESP,esp);
__SEIZEbits32(cpuModelGPR,EBX,ebx);
__SEIZEbits32(cpuModelGPR,EDX,edx);
__SEIZEbits32(cpuModelGPR,ECX,ecx);
__SEIZEbits32(cpuModelGPR,EAX,eax);
```

```
__SEIZEbits16(cpuModelSR,CS,cs);
__SEIZEbits16(cpuModelSR,DS,ds);
__SEIZEbits16(cpuModelSR,SS,ss);
__SEIZEbits16(cpuModelSR,ES,es);
```

```
inline bits16 cpuModelFR::Flags::operator = (const bits16 aux) {
  asm volatile ("push %0" : : "g" (aux));
  asm          ("popf");
  return aux;
}

inline cpuModelFR::Flags::operator bits16 () const {
  bits16 aux;
  asm          ("pushf");
  asm volatile ("pop   %0" : "=g" (aux));
  return aux;
}
```

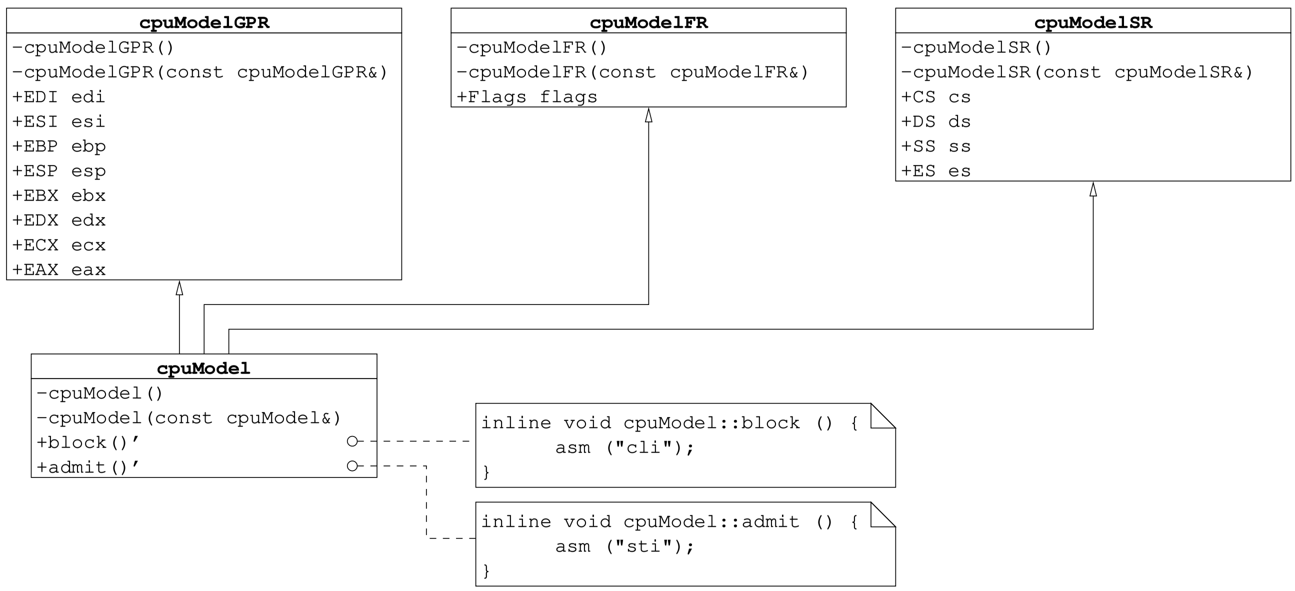
Register Access Functions — Exploitation

```
#include "cpu.h"

extern int foo ();

int main () {
  bits16 flags = cpu->flags;
  cpu->block();
  bits16 ds   = cpu->ds;
  cpu->ds     = cpu->cs;
  *(int*)foo = 0xC3;
  cpu->ds     = ds;
  cpu->flags = flags;

  cpu->eax    = 4711;
  return foo();
}
```

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

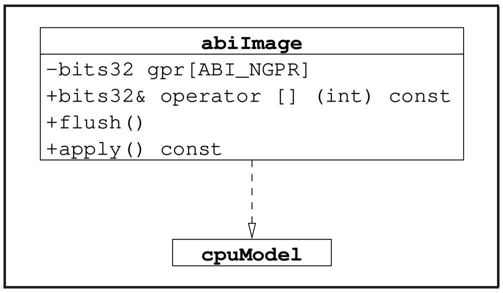
```
main:
  pushl %ebp
  movl  %esp,%ebp
  pushf
  pop   %cx
  cli
  mov   %ds,%dx
  mov   %cs,%ax
  mov   %ax,%ds
  movl  $195,foo__Fv
  mov   %dx,%ds
  push  %cx
  popf
  movl  $4711,%eax
  call  foo__Fv
  leave
  ret
```


Central Processing Unit — cpuModel (2)



State Buffering — ABI

- buffered save/restore of a thread's ABI state
  - RAF is used for CPU-register access
    - ∗ read/overwrite 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

State Buffering — abiImage

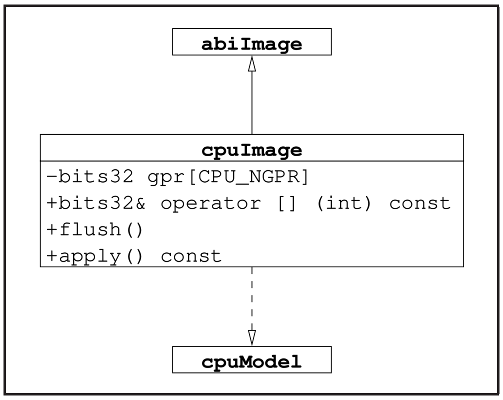
```
inline void abiImage::flush () {
  (*this)[ABI_EDI] = cpu->edi;
  (*this)[ABI_ESI] = cpu->esi;
  (*this)[ABI_EBP] = cpu->ebp;
  (*this)[ABI_EBX] = cpu->ebx;
}
```

```
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 — CPU

- 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

State Buffering — cpuImage (1)

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

```
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];
}
```

```
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 — `cpuImage` (2)

```
void foobar () {
  ...
  cpuImage foo;
  foo.flush();
  ...
}
```

```
...
movl %eax,-4(%ebp)
movl %ecx,-8(%ebp)
movl %edx,-12(%ebp)
movl %ebx,-16(%ebp)
movl %esp,-20(%ebp)
movl %ebp,-24(%ebp)
movl %esi,-28(%ebp)
movl %edi,-32(%ebp)
...
```

```
cpuImage bar;
void foobar () {
  ...
  foo[CPU_EAX] = foo[CPU_EBP] - foo[CPU_ESP];
  bar = foo;
  ...
}
```

```
...
movl   $bar,%edi
leal   -32(%ebp),%esi
...
movl   -20(%ebp),%eax
movl   -24(%ebp),%ecx
subl   %eax,%ecx
movl   %ecx,-4(%ebp)
cld
movl   $8,%ecx
rep
movsl
...
```

```
cpuImage bar;
void foobar () {
  ...
  bar.apply();
  ...
}
```

```
...
movl bar,%edi
movl bar+4,%esi
movl bar+8,%ebp
movl bar+12,%esp
movl bar+16,%ebx
movl bar+20,%edx
movl bar+24,%ecx
movl bar+28,%eax
...
```


Thread Interface Extension — `tie`

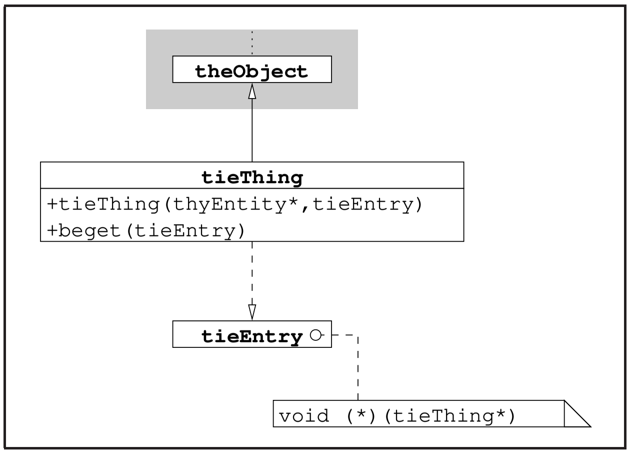
- logical separation of the thread abstraction(s) from the user abstraction(s)
  - encapsulation of thread instantiation and activation by a single method
- different ways of interfacing are provided by different (minimal) extensions

$$\text{to make the user thread a} \left\{ \begin{array}{l} \textit{pointer to} \left\{ \begin{array}{l} \textit{parametrized} \\ \textit{member} \end{array} \right\} \\ \textit{virtual} \\ \textit{default} \end{array} \right\} \textit{function}$$

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

Thread Interface Extension `tieThing`

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



- ordinary C functions become threads:

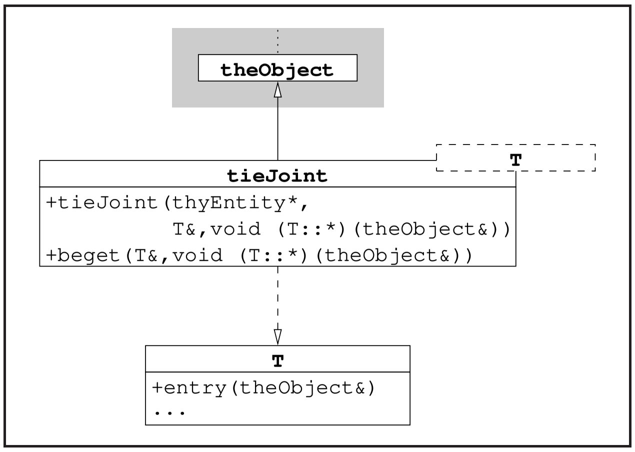
```
void gadget (tieThing* self) {
  self->grant(...);
}
```

- member functions of derived (single-inheritance path) classes may work also[4]

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

Thread Interface Extension `tieJoint`

- *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:

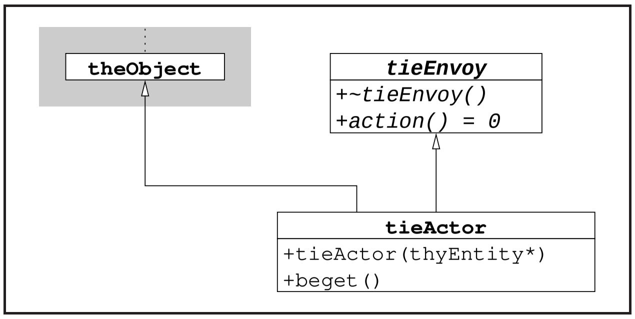
```
void T::entry (theObject& self) {
  self.grant(this->next);
}
```

- 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

```
void Foo::action () {
  grant(this->next);
}
```

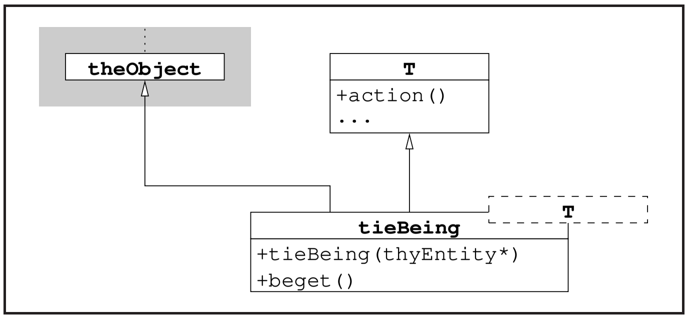


- a user thread is considered the final specialization of the system's abstraction(s)
  - it extends the system thread by user-defined attributes and methods
- the price to be payed is the presence of (at least) two virtual-function tables

Thread Interface Extension `tieBeing`

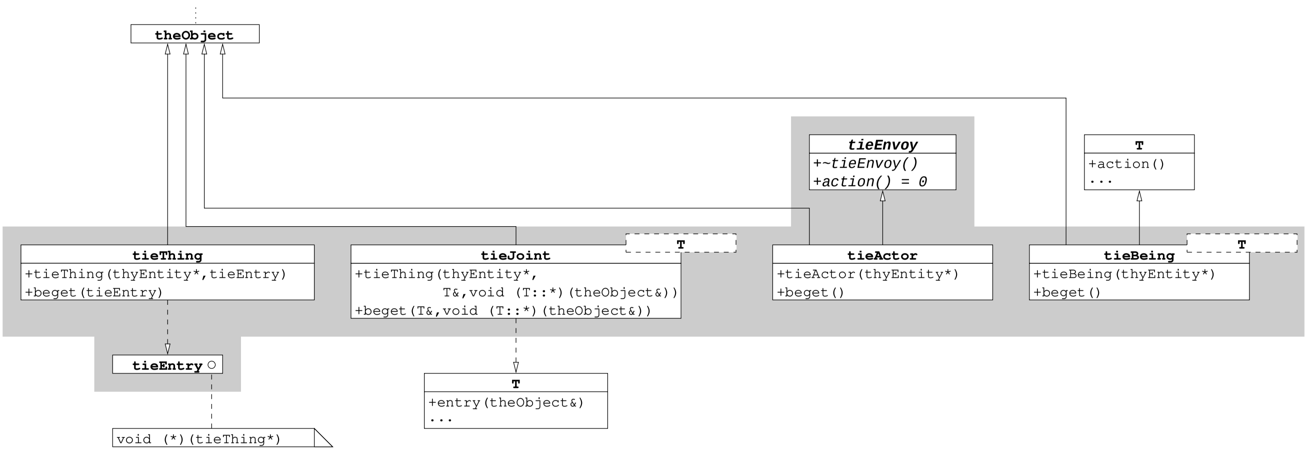
- *default function*
  - member of a user-defined class

```
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)

Thread Interface Extension — Class Hierarchy



Thread Abstraction Layer — Class Hierarchy