TAL — Feature Model

E

E

E

OSE

O

OE
aim at providing a minimal subset of thread functions

- compiler dependencies
- CPU management
- low-level scheduling
- control flows

design fundamental abstractions related to the following topics:

Thread Abstraction Layer — TAL
A thread's context is made of the contents of its processor registers.

- Stack-based context-saving shall come true:
  1. Stack all CPU registers
  2. Stack non-volatile CPU registers

The system description must contain at least one of these variants or feature:

- Stack-based context-saving shall come true:
  1. Stack all CPU registers
  2. Stack non-volatile CPU registers

- It needs to be temporarily saved during phases of thread inactivity.
- It depends on the CPU and the compiler/programming language.

Context Saving
Control Flow Invocation

- upon completion, the processor context is inherited to the resumed thread.
- thus thread invocation only implies exchanging the stack.
- the only private piece of context of the thread is its stack.
- inherit the (total) processor context to the spawned thread.
- upon completion, the spawned thread terminates and releases control.
- the spawned thread preempts the spawning thread, it runs until completion.
- allow the spawning of a thread, "inline" at (almost) any point of execution.
- provide a thread concept basing on an asymmetric invocation mechanism.
extend the thread concept by means of a symmetric invocation mechanism •

resumption happens "inline", at any point of execution •

thus thread resumption only implies a stack change

the threads all share the same processor context (← p. 3)

exchange

resumption happens on the thread's run-time stack — the thread's resumption point will be saved on the thread's run-time stack

a control-releasing thread remembers its resumption point (return address)

exchanging the flows of control happens in a coroutine-like fashion

Control Flow Exchange
threads need to save/restore their contexts by themselves

- requires the context-saving feature (p. 2)

provide thread switching in two ways:

- context saving (restoring) happens before (after) a resumption point

extend the thread concept by a coroutine style of context switching
Types of Coroutines

Independent Coroutines vs Dependent Coroutines

- Resume next thread
- Save thread's context
- Restore this thread's context
extend the thread concept to support the binding of user-defined code

three bindings are nice to have:

1. any sort of code fragment
2. pointer to function
3. specialized (virtual) function

optional features

aims at "user-friendliness"

any or none of them is valid

Code Binding
Stack Space Supply

The feature is nice to have, yet it’s non-functional and thus could be void.

An abstraction that should be optional, nevertheless purpose is to somehow provide means for “user-friendliness”.

- a (programming-language) representation as a template class, e.g.
- a specialization of the fundamental thread abstraction (← p. 3)
- provide a concept for compile-time stack-space allocation for a thread.
Threading Concept
Scheduling

the threading concept already provides a limited form of thread scheduling

CPU protection and other more enhanced policies are subjects for specialization

there is no (central) system-level thread scheduler

they keep on executing until control over the CPU is relinquished explicitly

there is no (central) system-level thread scheduler

threads are expected to be scheduled cooperatively under the user’s control

the threading concept already provides a limited form of thread scheduling
Scheduling Concept
A processor type can be implemented from scratch, encompassing assembly language programming to some extent, resulting in a "native implementation" running on the bare machine hardware abstractions are required for abstract processors, the programming language provides a thread concept, hiding processor peculiarities. Concrete processors, the thread concept must be implemented from scratch, depending on the capabilities of the underlying processor.

- Enabling portability at user level
- Hiding processor peculiarities
- Processor as an alternative feature
Register access

- assignment operators and type casts as in C++, e.g.:

\[
\text{means of operator overloading would be nice to have}
\]

- achieving portability is not the purpose at this level of abstraction

- a measure to improve the handling of processor-dependent stuff

- made feasible e.g., using "inline assembler" and/or `asm()` statements

- reading and writing of registers of the CPU’s programming model

- abstractions to access the CPU’s registers and processor-state management

Register Access
but there is the need to distinguish between different sorts of registers

State Buffering

State Buffering

State Buffering
The register set considered depends on both the concrete and abstract processor.

- sets of or-features
  - register banks
    - abstract
      - floating-point
        - special purpose
        - general purpose
      - concrete
  - the register set considered depends on both the concrete and abstract processor

Sorts of Registers
CPU Concept
being compiler-independent may be as difficult as being hardware-independent.

- "e.g. Register access using inline assembler (\textit{p. 13})"

  GNU gcc

being compiler-independent may be as difficult as being hardware-independent.

- "e.g. Register access using inline assembler (\textit{p. 13})"

programs may depend on the capabilities of that processor.

- "similar to differentiating among concrete processors (\textit{p. 12})"

software should differentiate between the various compiler releases.

- sooner or later, both typically undergo more or less major revisions.

Programming language and compiler implement an abstract processor.

Compiler
Compiler Concept
In general, of course this should hold for any kind of software design. Domain-analysis quality largely depends on the analyst’s domain experience.

- Processor type
- Compiler concept
- Non-functional requirements
  - CPU concept (except processor type)
  - Threading concept, scheduling concept
- Functional requirements

Summary