Operating-System Engineering

TAL — Modularization and Hiearchy

E

OSE

O
design the minimal subset of thread functions as a program family

by means of C-like code and its mapping to assembly-language level

by breaking down possible approaches for implementation

provide an experimental feasibility study of selected system functions

devlop the functional hierarchy of system abstractions to support threading.

Thread Abstraction Layer — TAL
The First Step is the Hardest

- the switching primitive's sole task is to swap the stack pointer's contents
  - invoked by the spawner (i.e., the creating thread)
  - finished by the spawnee (i.e., the spawned thread)
- as a by-product, the instantiation primitive will be entered once and left twice
  - by invoking the spawner's thread
  - by the spawner (i.e., the creating thread)
  - not by restoring the processor state of the involved threads.

Switching should mean to finish and resume program execution without saving
- of having activated a different runtime stack "on the fly".
- instantiation should mean to proceed program execution with the side-effect

Support inline instantiation of and switching between the threads.
Level 1

Flyweight Threads

**Latch** performs the termination of the current thread of control, i.e., it resumes the instructions immediately following.

Execution of the resumed thread continues at the "frozen" resumption address.

*Typically*, that bit pattern delivers a handle for the resumption of thread execution. Execution of another thread without leaving any resumption address behind.

Execution continues in place with the instruction used one. Execution continues in place with a different stack.

**Split** performs the instantiation of a new thread of control, i.e., it freezes the runtime stack address of the current thread of control.

**Label** delivers a bit pattern which is unique to the current thread of control and serves as a handle for the resumption of thread execution.

**Flyweight Thread** split performs the instantiation of a new thread of control, i.e., it (1) freezes the resumption address of the current thread of control, (2) fades in a runtime stack different from the currently used one, and (3) fades in a runtime stack address of the current thread of control.
FlyweightThreads (C-like)

... no, spawner resumed execution //
... yes, spawner started execution //
if slot = label() { // did a runtime-stack switch occur?
    spawn additional thread of control //
    if slot = label() { // remember current thread of control
        ... // yes, spawner finished and resumes spawner
        latch(slot); // spawner never returns to here
    } else { // no, spawner resumed execution
        ... // yes, spawner started execution
        latch(slot); // spawner never returns to here
    }
Flyweight Threads (x86)

```assembly
; Resume spawner
  leal-4(%esp),%edx
  #
  # slot=label()
  pushl$1f
  #
  # split(flux)
  movlflux,%esp
  #
  # "now spawner!!"
  leal-4(%esp),%eax
  #
  # .....=label()
  .... = label()
  #
  # slot=label()
  cmpl%eax,%edx
  #
  # "I: ... ...
  jf 2f
  #
  # 2f ...
  jf 2f
  #
  # "I: ...
  movl%edx,%esp
  #
  # latch(slot)
  ret
  #
  # "I: ...
  movl%esp,%edx
  #
  # resumption address
  cmp$tift,(fllzax)
  #
  # spawner end
  cmpl-4(%esp),%eax
  #
  # "I: ...
  ret
  #
  # resumption address
  call $4(%esp),%eax
  #
  # "I: ...
  pushl$fllzax
  #
  # spawner end
  cmpl-4(%esp),%edx
  #
  # "I: ...
  ret
  #
  # resumption address
  leal-4(%esp),%edx
```
spawn instantiates a new thread by exploiting label and split. Two threads will return from this function, at first the spawnee (non-zero return value) and then the spawner (zero return value). For the spawnee, the non-zero return value is the handle to later resume spawner execution.

shift transfers control to a thread different from the currently executing thread. The address of the stack location containing the resumption address of the control releasing thread will be saved for later purposes to resume that thread.

Control transfer is done by exploiting latch.
the goal can be met by assisting level 2 with (lower-level) support functions. •

Introducing replace to produce an assembly-language label (i.e., symbol):
•

Introducing check to encapsulate the assembly-language CPU instructions:
•

Introducing latch to execute of the next thread:
•

1. deliver and store the reference to the saved resumption address:
•

2. latch execution of the next thread:
•

3. provide a measure to support the generation of the resumption address:
•

An in-depth analysis of shift reveals three fundamental steps of execution:
•

- but not necessarily independent of an abstract "C/C++ processor", e.g.

Goal is to let the implementation of shift become independent of the CPU:
•

Control-Transfer of Featherweight Threads
Featherweight Threads (C-like)

```c
spawn (flux) {
    badge () ;
    latch (next) ;
    self = check () ;
}
```

```c
shift (self, next) {
    return slot != label () ? slot : 0 ;
}
```

```c
self = check () ;
```

```c
frozen slot thread //
resume next thread //
freeze this thread //
```
0 | \text{slot} = \text{aux} #
0 | 1 - \text{aux} #
1 | 0 = \text{aux} #
\vdots | \text{slot} == \text{aux} #
\text{zero aux} #
\text{badge()} #

\text{now spawnme!} #

\text{splice}(\text{flux}) #
\text{slot = tablet()} #

\text{Featherweight Threads Instantiation (x86)}
Featherweight Threads Resumption (x86)

```asm
shift (self, next) {
    pushl $1f
    movl %esp, (self)
    movl next, %esp
    ret
}

# badge()
# latch(next)
# self = .....
# ... = check()
resume
```

Operating-System Engineering — TAL — Modularization and Hierarchy
Featherweight Threads Exploitation (C-like)
both functions thus will constitute the (new) lowest level in the hierarchy.

- Higher-level (i.e., level 1 and 2) functions depend on this knowledge. Setup and badge share common knowledge about the resumption address. This functional commonality is worth to be abstracted by a dedicated function.

- Introducing setup to encapsulate the assembly-language CPU instructions.

- Generation and saving of the resumption address of the current thread: a further analysis of split and check reveals the following commonality.
where the resumption address of the current thread of control was saved.

check performs setup and delivers the address of the runtime-stack location to where the resumption address of the current thread of control was saved.

Support Function Level 1

setup generates a resumption address and places the computed value on the runtime-stack of the executing thread. The address is generated from a symbolic label (i.e., ‘label’) behind in the (assembly-language) code to

‘badge’ leaves a symbolic (i.e., label) behind in the (assembly-language) code to

setup generates a resumption address and places the computed value on the runtime-stack of the executing thread. The address is generated from a symbolic label (i.e., ‘label’) behind in the (assembly-language) code to

‘badge’ leaves a symbolic (i.e., label) behind in the (assembly-language) code to

Support Function Level 1

setup generates a resumption address and places the computed value on the runtime-stack of the executing thread. The address is generated from a symbolic label (i.e., ‘label’) behind in the (assembly-language) code to

‘badge’ leaves a symbolic (i.e., label) behind in the (assembly-language) code to
It seems as if there is a good chance that only `setup` (in addition to `latch`) becomes dependent on the CPU, i.e., needs to be hand-coded using assembly-language CPU instructions, rather than being dependent on the CPU's instruction set. This is because

```c
setup();
return cpu->sp;
```

where `latch` could be

```c
badge();
```

In the case of the `x86` architecture,
Support Functions (ppc, m68k, sparc)

```c
/* sparc */ {
  st [sp], 00%
  or [0], 00%
  sethi [0], 00%
  add [%sp], 4, [%sp]
} setup

/* ppc */ {
  stw 3, 0(1)
  la 3, 1(3)
  lis 3, 1(3)
  add 1, 1, 4
} setup
```
Level 3

Runtime-Stack Exploration

Depending on whether the registers of the abstract or the concrete processor are concerned, store and clear need to be realized in different versions.

store saves the contents of CPU registers onto the runtime stack of the currently executing thread. The stack will be extended by the amount of registers stored.

clear restores the contents of CPU registers from the runtime stack of the currently executing thread. The stack will be extended by the amount of registers stored.

top returns the initial value of the contents of the stack-pointer register given the base address and size of a stack segment, taking care of alignment restrictions.

clear restores the contents of CPU registers from the runtime stack of the currently executing thread. The stack will be cut back by the amount of registered cleared.

Depending on whether the registers of the abstract or the concrete processor are concerned, store and clear need to be realized in different versions.
Runtime-Stack Exploitation (x86)

Save and restore of the non-volatile general-purpose registers as defined by the application binary interface (ABI) of the compiler.

Save and restore of all general-purpose registers as defined by the programming model of the CPU.

```c
{ 
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    pushl %ebx
    movl %ebp, %esi
    movl %ebp, %edi
    movl %ebp, %ebx
    { 
        pushl %edi
        pushl %esi
        pushl %ebx
        pushl %ebp
        pushl %ebx
    } clear
    pushl % edi
    pushl % esi
    pushl % ebx
    pushl % ebp
    { 
        popl % edi
        popl % esi
        popl % ebx
        popl % ebp
    } store
}
```
new/deletereatesoperatorsofaclassusedtomodelflyweightthreads.

The typical implementation of both functions is (in C++) as overloaded

\[
\text{Stack-Pointer} \begin{aligned}
\text{new} & \quad \text{allocates a stack pointer by exploiting top with base address and size (in bytes)} \\
\text{delete} & \quad \text{deallocates a stack pointer virtually. Since new does not really result in a stack pointer, delete deallocates a stack segment referred to by some user-defined data type that goes conform rather to support the generation of a typed stack pointer that goes conform byties) of a runtime-stack segment. The purpose is not to allocate memory but}
\end{aligned}
\]

Operating-System Engineering — T.A.L. — Modularization and Hiearchy
Stack-Pointer

f

deg Allocation (C-like)

new(size, pool)

f

return top(pool, size);

g

new[](size, pool)

f

return top(pool, size);

g

delete(item)

f

assert(item == 0);

g

delte(item)

f

top(base, size)

{ /* 80C51 */

stack grows upward!

return base;

}

top(base, size)

f

{ /* PPC */

return base + size + ~(wordsize - 1);

}

top(base, size)

f

{ /* x86 */

return base + size;

}

top(base, size)

f

{ stack pointer (C-like) } De{ Allocation (C-like) } Stack-Pointer

}
There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.

There are as many control transfer functions as pairs of context-saving functions.

Accordingly, a thread itself is responsible to save and restore its context. Store and clear pair (\( p \leftarrow T \)) for saving and restoring the thread state. Both functions exploit shift to perform the control transfer and the respective store and clear pair (\( p \rightarrow T \)) for saving and restoring the thread state accordingly. A thread itself is responsible to save and restore its context.
Lightweight Threads (C-like)

{  
  clear()  
  shift(next, self)  
  transfer control  
  store()  
  grant(next)  
}

{  
  clear()  
  shift(next, self)  
  transfer control  
  store()  
  yield(next)  

restore full volatile registers  
restore non-volatile registers  
save full volatile registers  
save non-volatile registers
}
Lightweight Threads (x86)

(1)

yield (self, next)

{ popal #
  ret #
  movl %esp, (self) #
  movl %esp, %esp (self) #
  push $1 #
  pushal #
  yield (self, next) #
}

store() #

shift(self, next) #

movlnext, %esp #

movl %esp, (self) #

push #

ret #

store() #

clear() #
grant (self, next)

store()

shift (self, next)

clear()

{ ret
  # popl %eax
  # popl %ebp
  # popl %esi
  # popl %edi
  # pushl %ebp
  # pushl %esi
  # pushl %edi
  # pushl $1f
  # shift (self, next)
  # store ()
  # grant (self, next)
}

pushl %esp,(%edx)
# movl %esp,%eax
# ret
Level 5

- so far, users are concerned with all the peculiarities of the threading concept
- they are enabled to develop highly efficient multithreaded programs
- they are "obliged" to understand numerous design decisions
- separation of concerns implies to divide user code from threading code
  - i.e. to represent the user code e.g. as a
    - default function
    - pointer to function
    - pointer to member function
    - virtual method
  - the actual representation depends on the programming paradigm involved
Lightweight-Thread Instantiation

beget creates a new thread of control by exploiting (1) spawn to instantiate the thread, (2) yield to inherit the contents of the spawner’s general-purpose CPU registers to the spawnee, and (3) to assign user-defined code to the newly created thread.

The user-defined code is represented by an appropriate user-function abstraction (UFA). There are as many beget variants as UFA variants.

The user-defined code starts execution after having been explicitly enabled by the creator using either of the control-transfer functions: latch, shift, yield, or grant.
Lightweight-Thread Instantiation (C-like)

/* ptr. to member function */

void (*hook)(this)
{
    for (;;)
    {
        (*hook)(this);
    }
}

Operating-System Engineering — TAL — Modularization and Hierarchy
Everlasting Lifespan

by exploiting \texttt{beget}, the created thread is "condemned" to execute forever.

Also note that at the level of abstraction \texttt{beget} is assigned to a thread scheduler is still unknown. So there is no way to automatically run another thread in case of thread termination.

The only way is to embed the procedure call inside an endless loop.

- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
- the only way is to embed the procedure call inside an endless loop
Minimal Subset of Interface Functions

- The design put forward does not force users to pay for unneeded functions.
- Layman's "maybe concerned only with a minimal interface consisting of:

  - minimal subset of interface functions
  - to better customize the thread concept to their individual needs
  - to benefit from a much more simpler and efficient threading concept
  - to better instantiate a (lightweight) thread
  - to instantiate a well-aligned stack pointer
  - to allocate a well-aligned stack pointer
  - to represent the user-defined code to be executed by the thread

  UFA
  Grant
  to instantiate a lightweight thread
  to transfer CPU control between the threads
  to benefit from a much more simpler and efficient threading concept
  to better customize the thread concept to their individual needs
  minimal subset of interface functions

- "Laymans" may be concerned only with a minimal interface consisting of:
  - minimal subset of interface functions
  - to better customize the thread concept to their individual needs
  - to benefit from a much more simpler and efficient threading concept
  - to better instantiate a (lightweight) thread
  - to allocate a well-aligned stack pointer
  - to represent the user-defined code to be executed by the thread

Experts' may choose from a larger set of interface functions.
incremental system design relies on the postponement of design decisions. The stepwise functional extension smoothly approaches applications. If being in doubt of whether or not to include a feature, better exclude it. There is no alternative to fine-grain modularization in systems design. Reflection of the design decisions met is an ongoing process during design. Structural complexity is reduced by (coarse-grained) open components, with the coarse-grained building blocks being of fine-grained structure. Not always are common functions considered “common” instantaneously. With the coarse-grained building blocks being of fine-grained structure, the coarse-grained building blocks being of fine-grained structure. A refinement of proceeded design decisions must always be kept in mind.

**Summary**