O

O S E

E

TAL — Modularization and Hierarchy

Operating-System Engineering

Thread Abstraction Layer — TAL

- develop the *functional hierarchy* of system abstractions to support threading:

- provide an experimental feasibility study of selected system functions
  - by breaking down possible approaches for implementation
  - by means of C-like code and its mapping to assembly-language level
- design the minimal subset of thread functions as a *program family*

The First Step is the Hardest

- support *inline* instantiation of and switching between the threads:

  instantiation should mean to proceed program execution with the side-effect of having activated a different runtime stack "on the fly".

  switching should mean to finish and resume program execution without saving or restoring the processor state of the involved threads.

- as a by-product, the instantiation primitive will be entered once and left twice
  - invoked by the spawner (i.e., the creating thread)
  - finished by the spawnee (i.e., the spawned thread) and the spawner
- the switching primitive's solely task is to swap the stack pointer's contents

Level 1 — Flyweight Threads

split performs the instantiation of a new thread of control, i.e., it (1) freezes the resumption address of the current thread of control and (2) fades in a runtime stack different from the currently used one. Execution continues in place with the instructions immediately following.

latch performs the termination of the current thread of control, i.e., it resumes the execution of another thread without leaving any resumption address behind. Execution of the resumed thread continues at the "frozen" resumption address.

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. Typically, that bit pattern represents a runtime-stack address.

Flyweight Threads (C-like)

```
slot = label();         // remember current thread of control
split(flux);            // spawn additional thread of control
if (slot != label()) {  // did a runtime-stack switch occur?
  ...                   // yes, spawnee started execution
  latch(slot);          // spawnee finishes and resumes spawner
}                       // spawnee never returns to here
...                     // no, spawner resumed execution
```


Flyweight Threads (x86)

```
   leal  -4(%esp),%edx          # slot = label()
   pushl $1f                    # split(flux)
   movl  flux,%esp              #      "      now spawnee!!
1:                              # resumption address
   leal  -4(%esp),%eax          # ...... = label()
   cmpl  %eax,%edx              # if (slot == ...)
   je    2f                     # goto 2
   ...                          # ...
   movl  %edx,%esp              # latch(slot)
   ret                          #      "     resume spawner
2: ...                          # ...
```


Level 2 Featherweight Threads

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 been done by exploiting **latch**.

Control-Transfer of Featherweight Threads

- goal is to let the implementation of **shift** become independent of the CPU
  - but not necessarily independent of an abstract "C/C++ processor", e.g.
- an in-depth analysis of **shift** reveals three fundamental steps of execution:
  1. deliver and store the reference to the saved resumption address
     - introducing **check** to encapsulate the assembly-language CPU instructions
  2. **latch** execution of the next thread $\surd$
  3. provide a measure to support the generation of the resumption address
     - introducing **badge** to produce an assembly-language label (i.e., symbol)
- the goal can be met by assisting level 2 with (lower-level) support functions

Featherweight Threads (C-like)

```
spawn (flux) {
  slot = label();                      // freeze spawner
  split(flux);                         // instantiate spawnee
  return slot != label() ? slot : 0;   // generate result
}

shift (self, next) {
  self = check();                      // freeze this thread
  latch(next);                         // resume next thread
  badge();                             // resumption point
}
```


Featherweight-Threads Instantiation (x86)

```
spawn (flux) {
  leal  -4(%esp),%ecx             # slot = label()
  pushl $1f                       # split(flux)
  movl  flux,%esp                 #      "        now spawnee!!
1:                                # badge()
  leal  -4(%esp),%edx             # .... = label()
  xorl  %eax,%eax                 # zero aux
  cmpl  %edx,%ecx                 # slot == .... ?
  sete  %al                       # aux =    0 | 1
  decl  %eax                      # aux =   -1 | 0
  andl  %ecx,%eax                 # aux = slot | 0
}
```


Featherweight-Threads Resumption (x86)

```
shift (self, next) {
   pushl $1f                      # .... = check()
   movl  %esp,(self)              # self = .......
   movl  next,%esp                # latch(next)
   ret                            #      "       resume
1:                                # badge()
}
```


Featherweight-Threads Exploitation (C-like)

```
...
if (dad = spawn(flux)) {   // instantiate/run spawnee
  shift(son, dad);         // transfer control to spawner
  latch(dad);              // resume spawner, terminate
}
shift(dad, son);           // transfer control to spawnee
...
```

Support Functions

- a further analysis of **split** and **check** reveals the following commonality:
  - generation and saving of the resumption address of the current thread
- this functional commonality is worth to be abstracted by a dedicated function
  - introducing **setup** to encapsulate the assembly-language CPU instructions
- **setup** and **badge** share common knowledge about the resumption address
  - higher-level (i.e., level 1 and 2) functions depend on this knowledge
- both functions thus will constitute the (new) lowest level in the hierarchy

Support Functions — Level $\frac{1}{2}$

setup generates a resumption address and places the computed value on the runtime stack of the executing thread. The address is generated from a symbol left behind by **badge**.

badge leaves a symbol (i.e., label) behind in the (assembly-language) code to symbolically encode the thread's resumption address. This symbol is to be exploited by **setup**.

Support Function — Level 1

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 Functions (C-like/x86)

```
setup() {
  pushl $1f
} /* x86 */
```

```
badge() {
  1:
}
```

```
check() {
  setup();
  return cpu->sp;
}
```

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.

Support Functions (ppc, m68k, sparc)

```
setup() {
  addi 1,1,-4
  lis  3,1f@ha
  la   3,1f@l(3)
  stw  3,0(1)
} /* ppc */
```

```
setup() {
  movl #1f,a7@-
} /* m68k */
```

```
setup() {
  add   %sp,-4,%sp
  sethi %hi(1f),%o0
  or    %o0,%lo(1f),%o0
  st    %o0,[%sp]
} /* sparc */
```

Level 3 — Runtime-Stack Exploitation

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 cut back by the amount of registered cleared.

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.

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)

```
store () {              clear () {
  pushal                  popal
}                       }
```

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

```
store () {              clear () {
  pushl %ebx              popl %edi
  pushl %ebp              popl %esi
  pushl %esi              popl %ebp
  pushl %edi              popl %ebx
}                       }
```

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

Level 4 — Stack-Pointer {,De}Allocation

new allocates a stack pointer by exploiting **top** with base address and size (in bytes) of a runtime-stack segment. The purpose is not to allocate memory but rather to support the generation of a *typed stack pointer* that goes conform with some user-defined data type.

delete deallocates a stack pointer virtually. Since **new** does not really result in the allocation of a memory segment, the purpose of **delete** at this level of abstraction is to trap the attempt to deallocate a stack segment referred to by a stack pointer.

The typical implementation of both functions is (in C++) as overloaded new/delete operators of a class used to model flyweight threads.

Stack-Pointer {,De}Allocation (C-like)

```
new (size, pool) {             top (base, size) {
  return top(pool, size);        return base + size;
}                              } /* x86 */

new[] (size, pool) {           top (base, size) {
  return top(pool, size);        return base + size & ~(wordsize - 1);
}                              } /* ppc */

delete (item) {                top (base, size) {
  assert(item == 0);             return base;  // stack grows upward!
}                              } /* 80C51 */
```

Level 5 — Lightweight Threads

yield transfers control to another thread by saving and restoring the contents of all general-purpose registers as defined by the CPU's programming model.

grant transfers control to another thread by saving and restoring the contents of the non-volatile registers as defined by the compiler's *application binary interface* (ABI).

Both functions exploit **shift** to perform the control transfer and the respective **store** and **clear** pair (→ p. 17) 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.

Lightweight Threads (C-like)

```
yield (next) {
  store();               // save full register set
  shift(self, next);     // transfer control
  clear();               // restore full register set
}

grant (next) {
  store();               // save non-volatile registers
  shift(self, next);     // transfer control
  clear();               // restore non-volatile registers
}
```


Lightweight Threads (x86) (1)

```
yield (self, next) {
  pushal                # store()
  pushl $1f             # shift(self, next)
  movl  %esp,(self)     #    "
  movl  next,%esp       #    "
  ret                   #    "
1:                      #    "
  popal                 # clear()
}
```


Lightweight Threads (x86) (2)

```
grant (self, next) {
  movl  self,%edx
  movl  next,%eax
  pushl %ebx            # store()
  pushl %ebp            #    "
  pushl %esi            #    "
  pushl %edi            #    "
  pushl $1f             # shift(self, next)
  movl  %esp,(%edx)     #    "
  movl  %eax,%esp       #    "
  ret                   #    "
1:                      #    "
  popl  %edi            # clear()
  popl  %esi            #    "
  popl  %ebp            #    "
  popl  %ebx            #    "
  ret
}
```

Level 5* User-Function Abstraction

- 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 −
- *seperation 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

Level 6 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)

```
beget (this, hook) {
  if (dad = spawn(flux)) {
    yield(son, dad);
    for (;;)
      (*hook)(this);
  }
} /* ptr. to function */
```

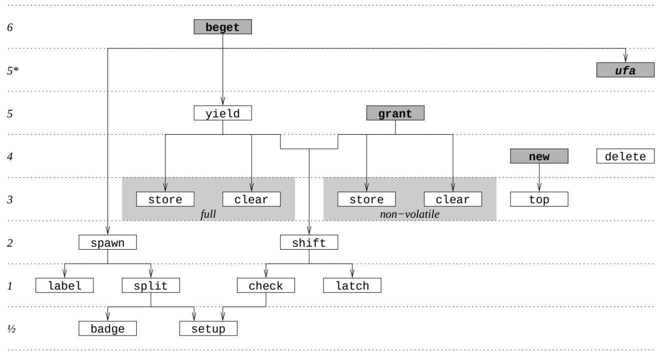
```
beget (this, hook) {
  if (dad = spawn(flux)) {
    yield(son, dad);
    for (;;)
      (this->*hook)();
  }
} /* ptr. to member function */
```


Everlasting Lifespan

- by exploiting **beget**, the created thread is "condemned" to execute forever
  - reason is the representation of the user-defined code as a procedure
    - from the user's viewpoint, thread termination equals procedure return
    - from the system's viewpoint, there is no idea to where to return to[1]
  - the only way is to embed the procedure call inside an endless loop
- there are two possible options to overcome the thread-termination problem:
  1. specialize and redefine **beget** once a scheduler has been designed, or
  2. provide for a "system UFA" (i.e., "wrapper") that solves the problem
- any way, the design decision on how to further proceed must be postponed

[1] Also note that at the level of abstraction **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.

Functional/Uses Hierarchy



Minimal Subset of Interface Functions

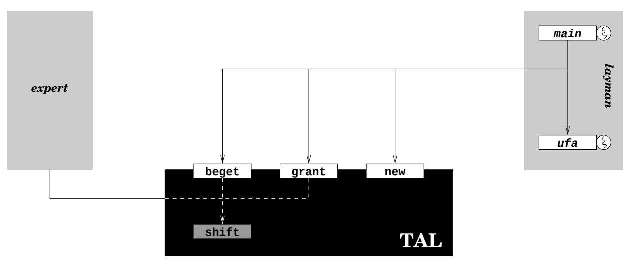
- "laymans" may be concerned only with a ***minimal interface*** consisting of:

  new to allocate a well-aligned stack pointer
  beget to instantiate a (lightweight) thread
  grant to transfer CPU control between the threads
  UFA to represent the user-defined code to be executed by the thread

- however, "experts" may choose from a larger set of interface functions
  - to benefit from a much more simpler and efficient threading concept
  - to better customize the thread concept to their individual needs

- the design put forward does not force users to pay for unneeded functions

Component View — Black-Box



Summary

- 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
- *reflection of the design decisions* met is an ongoing process during design
  - not always are common functions considered "common" instantaneously
  - a refinement of preceeded design decisions must always be kept in mind
- there is no alternative to *fine-grain modularization* in systems design
  - structural complexity is reduced by (coarse-grained) open components
  - with the coarse-grained building blocks being of fine-grained structure