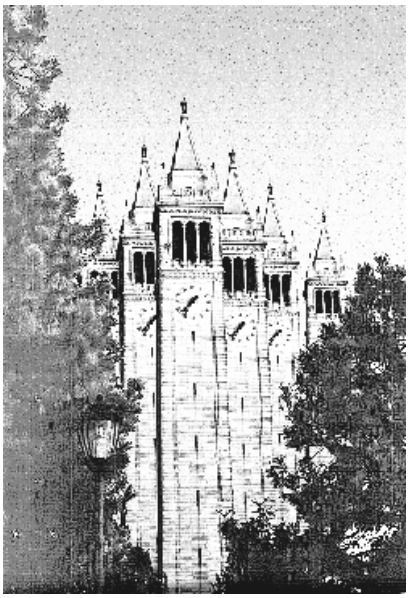

pSather 1.0

Killing Threads considered dangerous.

Claudio Fleiner

International Computer Science Institute, Berkeley

Joint work with the pSather group at ICSI:

Jerome Feldman
David Stoutamire
Boris Vaysman
Benedict Gomes
(and others)

pSather 1.0

- Overview of Sather and pSather
  - thread creation
  - synchronization
- Termination of Threads in pSather
  - different possibilities and their problems
- Threadsafe libraries

Sather 1.0

- object oriented language
- statically checked strong typing
- separate implementation and type inheritance
- portable and efficient (as fast as any other OOL)
- garbage collected
- exception handling
- parametrized classes
- iteration abstraction

pSather 1.0

adds support for parallel and distributed execution:

- threads
  multiple threads can execute in one object.
- data distribution,
  presents shared memory model to the programmer, even on distributed architectures.
- synchronization
  in the form of a flexible disjunctive lock statement.

Iterators

Iterators hide the code for initializing, updating, and testing iteration variables inside classes.

An iterator is defined like any standard routine, but instead of a `return` it has a `yield` to return successive values and a `quit` to end the loop it is called from.

Iterators
Example

The `elt!:T` iterator for arrays:

```
class ARRAY{T} is
    elt!:T is
        loop
            i::=0.upto!(size-1);
            yield self[i];
        end;
        quit;
    end;
```

Copying a container by using its `elt!` iterator:

```
    loop
        a.set!(b.elt!);
    end;
```

Creating Threads - Version 1

```
-- create new attach object
a:ATTACH:=#;


-- create some threads and
-- attach them to a
a :- function1;
a :- function2(arg);


-- wait until all threads attached
-- to a have finished
lock a.no_threads then
     -- no other thread can attach a new
     -- thread to a as long a is locked.
end;
```

Creating Threads - Version 2

```
par
    -- create new thread
    fork
        function1;
    end;
    -- create another thread
    fork
        function2(arg);
    end;
end;
-- the main thread continues only after
-- all threads created in the par
-- statements have finished.
```

Creating Threads - Version 3

```
parloop
   -- create a new thread for each element
   -- of array and execute frobnify for each
   el := array.elt!;
do
   el.frobnify;
done; -- wait for all threads
```

The parloop is used for data parallel applications.

Lock Statement Syntax

```
lock lck1, lck2, .. then
   statements;
end;

lock
[guard cond] when lck1, .. then
   statement1;
[guard cond] when lck2, .. then
   statement2;
   .
   .
[else
   statement_else]
end;
```

Lock Statement Properties

- Deadlock free

  All locks of a branch are locked atomically to avoid deadlocks of two threads trying to lock the same two locks at the same time.

- Guaranteed Unlock

  The system will unlock the locks as soon as the thread leaves the lock block, regardless if this happens because of a `return`*, a* `raise` *or a* `quit`*.*

- Fairness

  no starvation of threads or branches.

Synchronization Classes 1

- mutual exclusion lock

  `MUTEX`

  → *a single thread can lock the same object multiple times.*

- Reader/Writer locks

  `RW_LOCK:` readers have priority.
  `WR_LOCK:` writers have priority.
  `FRW_LOCK:` fair reader writer lock.

  → *a single thread can lock the same writer multiple times.*
  → *a thread can upgrade a writer to a reader, but not vice versa.*

Synchronization Classes 2

- Barrier locks
- Rendezvous locks
  threads can exchange information during the rendezvous.
- Condition locks

```
lock stack.not_empty then
    ...
end;
```

pSather provides an interface to the synchronization mechanism, which allows pSather users to create new synchronization classes in pSather without changing the runtime or the compiler.

Disjunctive Lock Example

Get an element from either `queue1` or `queue2`:

```
loop
   lock
      when queue1.not_empty then
         a:=queue1.pop;
      when queue2.not_empty then
         a:=queue2.pop;
   end;
   work_on(a);
end;
```

Asynchronous Thread Termination

A thread can respond to a termination signal

a) with immediate termination
breaks internal class structures.

b) by setting a flag
does not need special language support but does not work for stopped threads.

c) by executing a signal handler
useless unless the language provides a resume (longjmp).

d) by raising an exception

Raising an Exception

- catching the termination signal:

```
protect
    statements;
when TERMINATION then
    clean_up;
    raise TERMINATION;
end;
```

- blocking termination signal:

```
shield
    statements;
end;
```

Raising an Exception
Problem 1

The termination exception can happen at any time:

```
protect
   a:=#FILE("some_file");
when TERMINATION then
   -- we have no way to find out if the
   -- file has been opened or not
end;
```

Raising an Exception
Problem 2

Code reuse is harder, as the implementation of a routine may require a shield around the call:

```
--  multiply all elements of a vector
mult(m:INT) is
    loop set!(elt!*m); end;
end;
```

A termination in the middle of the loop leaves the vector in an inconsistent state.

The loop could be shielded, but this would slow down this basic operation.

Raising an Exception
Problem 3

How to pop a value from a stack without losing it:

```
done:=false;
protect
   lock stack.not_empty then
     shield
       e:=stack.pop;done:=true;
     end;
     work_on(e);
   end;
when TERMINATION then
    if done then
      stack.push(e);
    end;
end;
```

Raising an Exception
Problem 4

A thread that executes a bound routine and expects termination has no way to decide if it has to shield the call or not:

	may block forever	will not block forever
Routine expects termination	*call must not be shielded*	✓
does not expect termination	☠	*call must be shielded*

Raising an Exception
Problem 5

Many optimizations that change the order of execution are no longer possible:

```
protect
    a:=0;b:=0;
    a:=func1; -- returnvalue > 0
    b:=func2;
when TERMINATION then
    if a=0 then
        -- we know that b=0
    end;
end;
```

The compiler cannot change the order of the assignments in the protect block.

Raising an Exception
Conclusion

We concluded that raising an exception is the wrong way to stop a thread in an efficient language that promotes code reuse:

- it slows down libraries,
- defeats code reuse,
- makes some optimizations impossible,
- leads to clumsy code
- and introduces subtle bugs.

Terminating threads in pSather

A thread has to cooperate and

- poll a flag
  which works in most cases and can be done without special language support, unless the thread waits for a lock
- or use the disjunctive lock statement:

```
lock
when myMutex then
    critical_section;
when die then
    return;
end;
```

Consumer/Producer 1

```
producer(comm:QUEUE{INT}) is
--  enqueue 1..5 in the queue
   loop
      comm.enqueue(1.upto!(5));
   end;
end;

main is
   prod::=#ATTACH;
   queue::=#QUEUE;
   loop --  start 5 producer
      5.times!;
      prod :- producer(queue);
   end;
end;
```

Consumer/Producer 2

```
consumer(queue:QUEUE{INT},
         prod:ATTACH) is
--  dequeue until all producers died and
--  the queue is empty:
  e:INT;
  loop
    lock
    when queue.not_empty then
      e:=queue.dequeue;
    when queue.emtpy,
         prod.no_threads then
      return;
    end;
    work_on(e);
  end;
end;
```

Threadsafe Libraries

Each class has three types of routines (methods):

- **mutator** (`enqueue`)
  breaks/changes the internal class structures during execution. Only one mutator may be active in the object at any time.
- **visitor** (`head`)
  relies on, but does not change the internal class structures. Several visitors may run at the same time.
- **status checks** (`size`)
  does not rely on the class structure.

Visitor / Mutator Implementation

The visitor / mutator protocol is implemented with a reader / writer lock.

- A **mutator** locks the writer
- A **visitor** locks the reader
- a **status check** ignores the lock.

→ *As a writer lock can be upgraded to a reader, a mutator can call visitors, but a visitor cannot call a mutator.*

Visitor / Mutator Example: Queue

- mutators:
  `enqueue, dequeue, clear`

```
enqueue(e:T) is
    lock mutator then
        -- enqueue e
    end;
end;
```

- visitors: `head, tail, elt!`

```
head:T is
    lock visitor then
        -- return head
    end;
end;
```

- status checks: `size, is_empty`

```
is_empty is
    return size=0;
end;
```

Visitor / Mutator Iterators

- Definition:

```
elt!:T is
    -- this lock is released only after the
    -- loop that calls elt! has finished.
    lock visitor then
        loop
            i::=0.upto!(size-1);
            yield self[i];
        end;
    end;
end;
```

- Usage:

```
loop
    #OUT+array.elt!+'\n';
end;
```

- current state
  - The compiler has been around for 5 years now and runs on several architectures:
    - Network of Workstations, connected with TCP/IP or Myrinet
    - Meiko Supercomputer
    - Shared Memory Machines (Sun)
  - The lack of the termination signal has not been a problem.
- Future Work
  - Distributed Libraries
  - Optimizations
    - Lock & Visitor/Mutator protocol
    - memory/object distribution
    - parallelising / load balancing
  - Parallel Debugger
- Availability: *http://www.icsi.berkeley.edu/~sather*