Deferred Cancellation of Units of Execution

Philipp Bachmann
Institute for Medical Informatics and Biostatistics
Clarastrasse 12
CH–4058 Basel, BS
Switzerland

Revision: 1.7 *

October 4, 2008

Abstract
The pattern proposed helps to implement a portable termination and load adaptation mechanism for Active Objects [SSRB02a] and similar designs that delegate work to a pool of Units of Execution [MSM04, pp 217–221] to execute service requests asynchronously from their actual invocation.

For the pattern proposed we identified usage examples in popular existing applications or libraries.

An example accompanies the pattern presentation. This example shows sample code in C++.

A behavioral pattern

1 Intent
Safely shut down reactive systems without resource leakages.

2 Example
A webserver can deliver content to many clients in parallel. One implementation option to accomplish this is a pool of Units of Execution [MSM04, pp 217–221]—processes or threads—, each serving at most a single client at any one time. All Units of Execution run the same code, it’s the data that differs. The code basically consists of an infinite loop that contains a blocking system call to

*Copyright © 2008 Philipp Bachmann <bachlipp@web.de> and the Institute for Medical Informatics and Biostatistics. Permission is granted to copy for the 15th Pattern Languages of Programs (PLoP) conference 2008. All other rights reserved.
temporarily yield execution of a Unit until a client requests another webpage. During the implementation of a webserver it will turn out, that the provision for the startup of the Units is the least problem. It is much more difficult to develop a correct way to shut down the server again or to dynamically adapt the number of workers to varying load.

3   Context

Reactive systems, e.g. UNIX daemons or MS Windows services, are implemented as Active Objects [SSRB02a] or use the Half-Sync / Half-Async [SC96] [SSRB02b] or the Leader / Followers pattern [SSRB02c], [Ste98] pp 754–756]. These approaches are subsumed under the Worker Threads pattern [Lea99] pp 290–298], which in turn is a realization of the pattern Share the Load [Mes96] pp 588–589].

Monitor Objects [SSRB02d] and Reactors [SP, Sch95, SSRB02f] present blocking system calls to the Units of Execution within those reactive systems.

4   Problem

How to destroy an Active Object?

Active Objects and the similar concurrency mechanisms in Half-Sync / Half-Async or Leader / Followers implementations need to be shut down when the application receives a termination request. Additionally there might be the need to dynamically adapt the degree of concurrency within the reactive system to varying load, which means both increasing and decreasing the number of Units of Execution, the latter being a similar case to application termination.

Some programming languages and operating systems assist with the solution of the problem more than others (see e.g. Sections 10.3 and 10.4)—so particularly it is a challenge to portably destroy Active Object.

5   Forces

• Resource leaks are evil at runtime, because they accumulate. They are still ugly at termination, consider dangling lockfiles, files with the id of the (now gone) process, or still open connections to other systems, for example.

• The higher the level of abstraction, the better the portability.

• While blocking in a system call a Unit of Execution can not even terminate.

• Especially Units of Execution blocking in certain system calls are good candidates to cancel.
Table 1: Class–Responsibility–Collaboration Cards

<table>
<thead>
<tr>
<th>ActivationList</th>
<th>Hands service requests from</th>
<th>ActiveObject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>over to</td>
<td>PoolOfUnitsOfExecution</td>
</tr>
<tr>
<td></td>
<td>Throws</td>
<td>ActivationList-Disabled</td>
</tr>
</tbody>
</table>

(a) Activation List

<table>
<thead>
<tr>
<th>ActiveObject</th>
<th>Delegates service requests from</th>
<th>RemoteClients</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>to</td>
<td>PoolOfUnitsOfExecution</td>
</tr>
<tr>
<td></td>
<td>by means of</td>
<td>ActivationList</td>
</tr>
</tbody>
</table>

(c) Active Object

<table>
<thead>
<tr>
<th>PoolOfUnitsOfExecution</th>
<th>Takes service requests from</th>
<th>ActivationList</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>and executes them.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reacts upon</td>
<td>ActivationList-Disabled</td>
</tr>
</tbody>
</table>

(e) Pool of Units of Execution

<table>
<thead>
<tr>
<th>ActivationListDisabled</th>
<th>Thrown by</th>
<th>ActivationList</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(b) Activation List Disabled</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LocalClient</th>
<th>Creates</th>
<th>ActiveObject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Destroys</td>
<td>ActiveObject</td>
</tr>
</tbody>
</table>

(d) Local Client

<table>
<thead>
<tr>
<th>RemoteClients</th>
<th>Request services from</th>
<th>ActiveObject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>React upon</td>
<td>ActivationList-Disabled</td>
</tr>
</tbody>
</table>

(f) Remote Clients

6 Solution

Add a special exception class to your code. Identify those calls the concurrent Units of Execution may infinitely block in. Change the implementation of these calls to throw an instance of the exception class if and only if a certain flag is set. On termination set this flag and unblock the Units of Execution, both within the same Thread Safe Interface [SSRB02h] member function.

A first sketch of the solution is shown in Table 1. Figure 1 sketches the activity of Active Objects and their destruction. The design of the life cycle phases prior to destruction is not discussed in this pattern, so there are no activities related to these phases in the diagram. Section 11 refers to patterns that are concerned with such activities.

6.1 Participants

**ActivationList** Queues service requests to ActiveObject until there is a Unit of Execution from PoolOfUnitsOfExecution ready to finally execute the call. ActivationList implements the Monitor Object pattern, because it is shared among the Units of Execution of LocalClient and PoolOfUnitsOfExecution, and it uses synchronization depending on its State.
Figure 1: Activity diagram illustrating Active Object with Deferred Cancellation. For the sake of readability the concurrency inside was limited to one Unit of Execution.

\[ \text{ActivationList} \] implements a scheduling policy. Note that this participant does need to be explicit in case of Active Object and some Half-Sync / Half-Async server designs, but can be left implicit in case of Leader / Followers designs. ActivationLists can be disabled and indicate this by means of an ActivationListDisabled exception.

**ActivationListDisabled** Exception indicating not to expect any services from ActivationList.

**ActiveObject** Decouples a service request from its actual execution. So service requests are handled asynchronously. ActiveObjects delegate work to a PoolOfUnitsOfExecution it owns.\(^1\) It indirectly hands work over to the pool by means of the intermediate ActivationList.

**LocalClient** The LocalClient owns ActiveObject. First it creates an ActiveObject, later it destroys it.

**PoolOfUnitsOfExecution** Each Unit takes member function requests out of ActivationList to execute them on behalf of the ActiveObject. The Units handle ActivationListDisabled exceptions. A Unit is called Servant in the original Active Object pattern.

\(^1\)For the sake of this paper ActiveObjects include implementations of the Half-Sync / Half-Async and the Leader / Followers pattern, because regarding to cancellation all three are very similar to each other.
RemoteClients  RemoteClients request services from ActiveObject.

Figure 2 sketches the participants and their relations to each other.

6.2 Dynamics

At startup LocalClient creates ActiveObject, which in turn creates ActivationList and PoolOfUnitsOfExecution. The details were described in [SSRB02a, p 425] and [Bac05, pp 24–25].

This mechanism can be adapted to the case that by means of some metric LocalClient gets forced to increase the size of PoolOfUnitsOfExecution.

During the lifetime of ActiveObject RemoteClients send service requests to ActiveObject. ActiveObject reifies the requests as Commands [GHJV96a] and hands the Commands over to ActivationList, its Command Processor [BMR+00a]. This also was described in detail before. Here the patterns Active Objects, Half-Sync / Half-Async, and Leader / Followers differ from each other.

The termination of ActiveObject starts with a person or a parent process sending a termination signal to LocalClient. LocalClient in turn disables ActivationList. ActivationList sets a flag and wakes up all Units from PoolOfUnitsOfExecution blocking in a member function of ActivationList to receive new work. Before returning from the member function each Unit checks the flag set by the LocalClient Unit of Execution before. Because it is set, the member function is left now by means of throwing ActivationListDisabled. Each Unit is given the chance to release resources and then terminates. LocalClient then joins the Units of Execution, i.e. it waits for all of them to terminate. Now LocalClient releases its own resources including resources shared among PoolOfUnitsOfExecution and terminates.

The latter mechanism can be adapted to the case that by means of some metric LocalClient gets forced to reduce the size of PoolOfUnitsOfExecution.

The dynamics of Deferred Cancellation is shown in Figure 3.

6.3 Rationale

This pattern is about controlled cancellation of Units of Execution by another Unit of Execution.
The solution proposed provides for well defined cancellation points: The member functions of ActivationList throw a special exception if and only if it is disabled.

Each Unit from PoolOfUnitsOfExecution is not simply being forced to immediately die. It is given the chance to release resources.

The design proposed consists of joinable Units of Execution, not of detached ones. So LocalClient can wait until the last Unit from PoolOfUnitsOfExecution has terminated, before resources shared among the Units and itself are going to be released. If you go for detached Units of Execution instead, combine Thread Safe Interface with one of the following smart pointer patterns: Counted or Detached Counted Body idiom [Cop00, pp 173–179], [Lak96, pp 429–434], [Str98, pp 841–845] or Shared Ownership [Car96] to protect ActivationList and other shared resources from premature release (see Section 8.2 for an example).

7 Resulting Context

Blocking calls in PoolOfUnitsOfExecution were identified. There is a mechanism in place to transmit cancellation requests from LocalClient to the members of PoolOfUnitsOfExecution. All Units of Execution implement means to react upon cancellation requests by releasing resources and finally quitting.

7.1 Pros and Cons

The Asynchronous Cancellation pattern has the following benefits:

1. **Well-defined cancellation points.** The control flows of the working Units of Execution get interrupted at clearly defined calls.

2. **Opportunity granted to release resources.** After receiving a cancellation request the workers can release resources acquired before, be it memory, be it open files, be it any other resource.

3. **Portability.** This pattern works on all platforms that allow for handcrafting the Monitor Object pattern, the core of ActivationList.
4. Cooperation. As pointed out by Anthony Williams, one of the authors of the current Boost.Thread implementation, using exceptions to force termination gives Units from PoolOfUnitsOfExecution great flexibility to react upon such requests [Wil08].

The Asynchronous Cancellation pattern has the following liability:

1. Cooperation. Benefit can also turn into a liability. Each Unit from PoolOfUnitsOfExecution can react upon ActivationListDisabled by ignoring it. As LocalClient only terminates after all Units have terminated, a malicious Unit has the power to veto against termination.

Additionally to these general pros and cons we identified the following implementation specific ones.

The implementation technique of the Asynchronous Cancellation pattern shown has the following liability:

2. There might be too few cancellation points. In case of some Half-Sync / Half-Async or all Leader / Followers server designs the blocking system calls are select() (a Reactor) or accept(), which you have less control over than regarding to the condition variables aggregated by Monitor Objects, the core of ActivationList. A workaround is as follows: The controlling Unit of Execution sends a network packet to the port the application listens to. This unblocks the Leader. Then both the Leader and the Followers may be treated as described above.

8 Implementation

Depending on the platforms the application is required to work on an implementation might rely on features of the operating system to shut down (see Section 10). If this is not possible, you need a way to intercept blocking calls. The following two Sections conform to the description of the pattern in that it steps into blocking calls not at Wrapper Facades [SSRB02] of condition variables, the lowest possible layer of abstraction, but at the higher-level Monitor Object.

The code is presented as self-contained as possible. Therefore we had to decide on the nature of the Units of Execution: Here we consider threads. To allow for a quick identification of the core of this pattern the most important entities have been underlined, and a comparison between code artifacts and the participants is given in Section 8.3.

8.1 Example Resolved

The code in Listing 1 shows a very simple Active Object without sophisticated scheduling: Every service request gets executed as soon as there is a free Unit of Execution available. Only the cancellation aspect is shown in detail. Other aspects are discussed in [SSRB02a, p 425].
Listing 1: Cancellability added to scheduler from [SSRB02a p 425]

```c
extern "C" {
    void *svc_run(void *);
}

struct Method_Request {
    virtual ~ Method_Request();
    virtual void call() =0;
};

struct Message_Queue_Disabled {};

class Message_Queue {
    ...
    std::size_t max_messages_; 
    mutable Thread_Mutex monitor_lock_; 
    Thread.Condition not_empty_;  
    Thread.Condition not_full_;  
    volatile bool isActive_;  
public:
    enum { MAX_MESSAGES = ... }; 
    explicit Message_Queue(std::size_t max_messages =MAX_MESSAGES) :
        max_messages_(max_messages), 
        not_empty_(monitor_lock_), 
        not_full_(monitor_lock_), 
        isActive_(true), ... {
        ...
    }
    Message_Queue(const Message_Queue & rhs) :
        not_empty_(monitor_lock_), // Condition variables can't 
        not_full_(monitor_lock_), // be copied 
        isActive_(rhs.isActive), ... {
        ...
    }
    bool empty_i() const; 
    bool emptyAndEnabled_i() const {
        return empty_i() && isActive_; 
    }
    bool full_i() const; 
    bool fullAndEnabled_i() const {
        return full_i() && isActive_; 
    }
    void disable() {
        Thread_Mutex_Guard guard(monitor_lock_); 
        isActive_=false; 
        not_empty_.notify_all(); 
        not_full_.notify_all(); 
    }
    // Transfers ownership 
    Method_Request *get() {
        Method_Request *result=0;
        {
            Thread_Mutex_Guard 
            guard(std::move(not_empty_.wait_if( 
                boost::bind( 
                    &Message_Queue::emptyAndEnabled_i, 
                    this, 
                    boost::bind( 
                        &Message_Queue::fullAndEnabled_i, 
                        this), 
                    boost::bind( 
                        &Message_Queue::full_i, 
                        this), 
                    boost::bind( 
                        &Message_Queue::empty_i, 
                        this)))); 
        }
        return result; 
    }
};
```
8.1 Example Resolved

```cpp
    this,
    _1
  )
  )))
  if(!isActive_
    throw Message_Queue_Disabled;
  const bool wasFull(full_i());
  result=get_i();
  if(wasFull)
    not_full_.notify_all();
  }
  return result;
}
// Transfers ownership
void put(Method_Request *msg) {
  Thread_Mutex_Guard
  guard(std::move(not_full_.wait_if(
    boost::bind(
      &Message_Queue::fullAndEnabled_i ,
      this,
      _1
    )
  )
  )))
  if(!isActive_
    throw Message_Queue_Disabled;
  const bool wasEmpty(empty_i());
  put_i(msg);
  if(wasEmpty)
    not_empty_.notify_all();
};
class MQ_Scheduler {
  typedef std::vector< thread_type > pool_type;
  Message_Queue act_queue_; 
  pool_type pool_; 
  void joinPool_() {
    for(pool_type::reverse_iterator in(pool_.rbegin());
      pool_.rend()!=in;
      ++in)
      joinThread(*in);
  }
  public:
    MQ_Scheduler(std::size_t high_water_mark ,
      std::size_t number_of_threads) 
      : act_queue_(high_water_mark) {
    pool_.reserve(number_of_threads);
    try {
      for(std::size_t i(0); number_of_threads>i;++i)
        pool_.push_back(createThread(svc_run,&act_queue_));
    }
    catch(...) {
      act_queue_.disable();
      joinPool_();
      throw;
    }
};
```
IMPLEMENTATION

```cpp
MQ_Scheduler() {
    act_queue_.disable();
    joinPool_();
}

// Transfers ownership
void insert(Method_Request *method_request) {
    act_queue_.put(method_request);
}
}

void * svc_run(void * arg) {
    // Set all signals blocked in the current thread's signal mask
    ...
    assert(arg);
    Message_Queue *act_queue = static_cast<Message_Queue *>(arg);
    while (true)
        try {
            // Block until the queue is not empty
            std::auto_ptr<Method_Request> mr(act_queue->get());
            mr->call();
        } catch (const Message_Queue_Disabled &) {
            break;
        } catch (...) {
        }
    return 0;
}
```

Note some details of this implementation:

- `Message_Queue::disable()` uses Scoped Locking [SSRB02g] to acquire and unconditionally release the lock again, an application of Resource Acquisition is Initialization [Str98, pp 388–393], [Str94, pp 495–497].

- `Thread_Condition` and `Thread_Mutex` are Wrapper Facades that turn operating system specific imperative interfaces into object oriented ones without impacting performance. Wrapper Facades still give the compiler the opportunity to inline the respective member functions.

- `Message_Queue::{get(), put()}` make use of condition variables equipped with move semantics proposed for future revisions of the C++ standard [HSK06] to become able to return Scoped Locking guards by value from `Thread_Condition::wait_if<>()`, a Convenience Method [Hir97] similar to what Boost.Thread offers. Move semantics can be implemented today with help of the Change of Authority idiom [Bac05].

- `MQ_Scheduler::joinPool_()` is incomplete in that it does not call `pool_.clear()`. It is an implementation helper only and is therefore declared private.

- The templated constructor calls `std::vector<>::reserve()` to prevent `std::vector<>::push_back()` to throw an exception.
8.1 Example Resolved

- `thread_type`, `createThread()`, and `joinThread()` wrap the respective operating system specific types and functions. `createThread()` translates error codes into exceptions.

  From a design point of view `thread_type` is a Future type, and the role of the Rendezvous function is taken by `joinThread()` [SSRB02a, pp 413, 417, 423–430, 435–436]. `MQ_Scheduler::pool_` is an instance of a Future container, `MQ_Scheduler::joinPool_()` being its Rendezvous function.

- The main loop of the threads in `svc_run()` is in fact a simple example of the Leader / Followers design pattern: The threads line up to become a Leader, i.e. they are Followers. Only one thread at any one time can take new work out of the queue, i.e. it takes the role of the Leader. On returning from `Message_Queue::get()` `Message_Queue::monitor_lock_` is being released, thus implicitly a new Leader gets designated. The former Leaders can process their work packages concurrently then.

Listing [2] shows how the parts proposed above work together. No emphasis was put on where the service requests actually originate from, i.e. on an useful interface to `RemoteClient`.

Listing 2: How to use `MQ_Scheduler`

```cpp
namespace {
    std::jmp_buf env_;
} /* (anonymous) namespace */

extern "C" {
    static void SIGTERMDisposition_(int sig) {
        longjmp(env_,1);
    }
}

int main() {
    signal(SIGTERM,SIGTERMDisposition_);
    MQ_Scheduler sched;
    std::auto_ptr<Method_Request> mr;
    if(!setjmp(env_)) {
        while(true) {
            // Very rough sketch...
            mr.reset(new Concrete_Method_Request);
            sched.insert(mr.release());
        }
    }
    return EXIT_SUCCESS;
} // On automatic destruction of "sched" first its workers
    // will terminate after having caught a "Message_Queue_Disabled"
    // exception. Then they are going to be joined before
    // "Message_Queue_Disabled::act_queue_" is destructed.
    // After all, "MQ_Scheduler" vanishes.
```

In an even simpler setting the main thread can simply block in `pause()` until a signal comes in as described e.g. in [Ste98] instead of the above signal handler and its interaction via `longjmp()` and `setjmp()` that have to be used with
8 IMPLEMENTATION

Figure 4: Class diagram matching Listings 1 and 2.

caution to avoid resource leaks. \texttt{mr} was defined outside the \texttt{setjmp()} block to avoid it not to be destructed in case of termination.

Figure 4 sketches the participants and their relations to each other. The dynamics of the entities in this example is shown in Figure 5. Note that \texttt{Thread\_Condition} automatically deactivates the associated \texttt{Thread\_Mutex} on blocking and activates it again before returning in \texttt{Thread\_Condition::wait()}.

8.2 Dynamic Adaptation to Varying Load

This section extends the example with a mechanism to dynamically shut down some Units of Execution to adapt their number to decreasing load. Reasons for the need for this dynamic adaptation are e.g.

- Threads consume resources even if they block, so they should be limited in number to the expected load.
- Many reactive systems are I/O bound, so the performance can benefit from more threads than there are processor cores.
- It is hard to estimate the load up-front. Furthermore, the load likely varies.

From a more abstract point of view Half-Sync / Half-Async server designs turn asynchrony into synchrony, which on the one hand yields a programming model easier to deal with than event-driven I/O strategies, but on the other hand results in the need for dynamic load adaptation.

Several aspects need to be refined in order to do so: We want to cancel some, but not all threads of \texttt{PoolOfUnitsOfExecution}. As all of these threads execute the same code, it is reasonable to assume that the code does not need to cancel a particular thread—all we need is a way to cancel a certain number
8.2 Dynamic Adaptation to Varying Load

Figure 5: Sequence diagram matching Listings 1 and 2. For the sake of readability the concurrency inside was limited to one thread.
IMPLEMENTATION

of threads from the pool regardless of their identity. So the cancellability of `Message_Queue::get()` got changed in a way that the number of threads to send the exception to can be specified.

Furthermore, we can now cancel getting work from `Message_Queue` independently from putting new work to the queue—only the first operation needs to be cancelled during dynamic adaptation to decreasing load, because the RemoteClients must not be affected by this internal operation.

The code in Listing 3 shows the modified `Message_Queue`.

Listing 3: Message queue with refined cancellability

```cpp
class Message_Queue {  
  ...
  volatile std::size_t consumers2Cancel_;  
  volatile bool isProducerActive_;  
public:
  explicit Message_Queue(std::size_t max_messages = MAX_MESSAGES)  
    : ...,  
      consumers2Cancel_(0),  
      isProducerActive_(true), ... {  
    ...
  ...
  bool emptyAndEnabled_i() const {  
    return empty_i() && 0==consumers2Cancel_;  
  }
  bool fullAndEnabled_i() const {  
    return full_i() && isProducerActive_;  
  }
  void disableGet(std::size_t consumers2Cancel) {  
    Thread_Mutex_Guard guard(monitor_lock_);  
    consumers2Cancel_=consumers2Cancel;  
    not_empty_.notify_all();  
  }
  void disablePut() {  
    Thread_Mutex_Guard guard(monitor_lock_);  
    isProducerActive_=false;  
    not_full_.notify_all();  
  }
  // Transfers ownership  
  Method_Request *get() {  
    Method_Request *result=0;  
    {  
      Thread_Mutex_Guard  
      guard(std::move(not_empty_.wait_if(  
        boost::bind(  
          &Message_Queue::emptyAndEnabled_i ,  
          this,  
          _1  
        )));  
      if(0!=consumers2Cancel_--)
```
8.2 Dynamic Adaptation to Varying Load

```cpp
throw Message_Queue_Disabled;
const bool wasFull(full_i());
result = get_i();
if (wasFull)
    not_full_.notify_all();
}
return result;

// Transfers ownership
void put(Method_Request *msg) {
    Thread_Mutex_Guard
    guard(std::move(not_full_.wait_if(
        boost::bind(
            &Message_Queue::fullAndEnabled_i,
            this,
            _1
        )
    )));
    if (!_isProducerActive_
        throw Message_Queue_Disabled;
    const bool wasEmpty(empty_i());
    put_i(msg);
    if (wasEmpty)
        not_empty_.notify_all();
}
};
```

Simply cancelling some of the threads from the pool regardless of their identity also means that it becomes difficult to both join the threads that have gone and to maintain a list of ids of still active threads later to join. Therefore we use detached threads here and emulate `joinThread()` by our own mechanism.

The heart of this is a threadsafe incarnation of the Detached Counted Body idiom [Cop00, pp 176–179], [Lak96, pp 429–434], [Str98, pp 841–845] or Shared Ownership [Car96] (see Page 6), shown in Listing 4.

Listing 4: Smart pointer that both protects shared data and works as a Future

```cpp
class Message_QueueHandle {
    class Count {
        // No copy allowed, therefore private and declared only
        Count(const Count &);
        // No assignment allowed, therefore private and declared only
        Count &operator=(const Count &);
        public:
            std::size_t count_;  // private
            mutable Thread_Mutex lock_;  // private
            Thread_Condition unique_;  // private
            Count() : count_(1), unique_(lock_) {}  // private
        }
    private:
    Message_Queue *rep_;  // private
    Count *count_;  // private
    public:
        explicit Message_QueueHandle(Message_Queue *rep)
            : rep_(rep), count_(new Count) {}  // private
        Message_QueueHandle(const Message_QueueHandle &rhs)
            : rep_(rhs.rep_), count_(rhs.count_) {  // private
            atomicIncrement();  // private
    }
};
```
Instances of this class protect an instance of Message_Queue from premature destruction. As shown below, Message_QueueHandle::waitUnlessUnique() makes instances of this class Futures.

The scheduler only needs slight adaptations, whereas the main thread loop now periodically tests whether to cancel or start threads—the respective metric is only sketched in the following code. Listing 5 shows the remaining code fraction.

Listing 5: Scheduler and thread main loop

```cpp
class MQ_Scheduler {
    Message_QueueHandle act_queue_;

    void joinPool_() {
        act_queue_.waitUnlessUnique();
    }

public:
    MQ_Scheduler(std::size_t high_water_mark,
                  std::size_t number_of_threads)
        : act_queue_(new Message_Queue(high_water_mark)) {
        std::size_t i(0);
        try {
            for(;number_of_threads>i;++i)
            {
                act_queue_.atomicIncrement();
                createDetachedThread(svc_run,&act_queue_);
            }
        } catch(...) {
            // Creation of (i+1)th thread failed to happen.
        }
    }
};
```
Additionally to the comments on the code in the section before note some details of this implementation:

- As the instance of `Message_QueueHandle` must be passed by pointer to `svc_run()` it is important to temporarily manipulate its reference counter.
- The main thread waits until it is the only one that holds a reference to
the contents of \texttt{Message\_QueueHandle}. To implement waiting, a condition variable is used within \texttt{Message\_QueueHandle}.

The usage of the modified scheduler does not differ from the first example.

### 8.3 Relationship of Examples and Participants

The code examples shown above map to the participants defined in Section 6.1 as follows.

<table>
<thead>
<tr>
<th>Code</th>
<th>Participant</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Message_Queue}</td>
<td>ActivationList</td>
</tr>
<tr>
<td>\texttt{Message_Queue_Disabled}</td>
<td>ActivationList_Disabled</td>
</tr>
<tr>
<td>MQ_Scheduler</td>
<td>ActiveObject</td>
</tr>
<tr>
<td>\texttt{main}</td>
<td>LocalClient</td>
</tr>
<tr>
<td>Section 8.1 MQ_Scheduler::pool; Section 8.2 Implicit.</td>
<td>PoolOfUnitsOfExecution</td>
</tr>
<tr>
<td>\texttt{main}</td>
<td>RemoteClients</td>
</tr>
</tbody>
</table>

### 9 Variants

The level of abstraction chosen for the interception of blocking calls can vary. So a variant is to equip condition variables with an operation to disable their operation and to let all their potentially blocking operations throw if and only if they have been disabled. If portability is an issue, then choosing a high a level as suggested can make life easier.

Another variant is to make cancellability a property of a Unit of Execution instead of a class with blocking operations.

### 10 Known Uses

Examples of Static Strategy can be found in existing software.

#### 10.1 ACE

The ADAPTIVE Communication Environment (ACE) provides \texttt{ACE\_Message\_Queue<>}, a synchronized queue, that can be disabled similar to the one proposed above. Its blocking member functions do not throw if the queue has been disabled. Instead they return the special value \texttt{ESHUTDOWN} then.
10.2 Boost.Thread

Deferred cancellation was recently added to the Boost.Thread [Kem02 Wil] C++ library. Cancellation is referred to as “interruption”. Different from the pattern description above it’s not ActivationList that can be disabled, but it is the thread itself. This is similar to Java (see Section 10.4). Interruption takes place by means of an exception thrown from well-defined interruption points, e.g. boost::condition_variable. The user can define additional interruption points, if necessary. The set of pre-defined interruption points is still limited to entities within Boost.Thread—there are many more blocking calls, however, e.g. in std::fstream or during network communications, because cancellability is a crosscutting concern [KLM+97].

Internally, Thread-Specific Storage [SSRB02i] is used to manage the flag that indicates interruption of a thread for it to be accessible from the interruption points.

10.3 POSIX Threads

POSIX 1003.1c compliant systems provide pthread_cancel() that cancels a thread specified as an argument. Threads can be configured dynamically to react differently upon cancellation requests: They can ignore them, they can be cancellable at well-defined cancellation points only (“deferred cancellation”), or they can immediately exit (“asynchronous cancellation”). In both of the last two cases during the cancellation procedure so-called cleanup handlers are going to be executed, that have been registered by the user before. The default behavior is cancellation enabled at well-defined cancellation points only. The set of POSIX Threads cancellation points is extensible. The list of cancellation points defined in UNIX98 lists accept() as a cancellation point, POSIX 1003.1c does not. For details see e.g. [But97].

Some C++ compilers automatically register destructors as cleanup handlers.

10.4 Java

Similar to Boost.Thread (see Section 10.2) instances of the class java.lang.Thread can be set interrupted. In this case some blocking operations are left with an instance of java.lang.InterruptedExcept thrown, some with an instance of java.nio.channels.ClosedByInterruptExcept thrown, while calls to java.nio.channels.Selectors are simply waked up. For details see e.g. [Lea99] pp 169–177,294).

10.5 MS Windows

MS Windows is listed here as a counterexample. TerminateThread() simply kills a thread immediately without giving it a chance to clean up.

Implementing Active Objects and thread pools with a mechanism that automatically adapts the size to varying load on MS Windows requires code similar
REFERENCES

to that of Section 8.1 that does not rely on safe cancellation provided by the operating system.

11 Related Idioms and Patterns

The context of Deferred Cancellation is formed by a pool of Units of Execution, e.g. Worker Threads. This is the basis of Active Objects, Half-Sync / Half-Async or Leader / Followers designs.

Deferred Cancellation shows how to safely clean up resources specific to members of PoolOfUnitsOfExecution and shared among many Units. These resources must have been initialized before. Kircher / Jain have a comprehensive collection of initialization patterns [KJ04, pp 19–79].

The mechanism to force cancellation of the members of the pool steps in at Wrapper Facades or higher level abstractions like Monitor Objects.

If static configuration of the platforms is required or there is the need to choose among different higher–level architectures, e.g. between the designs Leader / Followers, One Child per Client [Ste98, pp 732–736] and One Thread per Client [Ste98, pp 752–753], then static and metaprogramming patterns—especially Static Adapter and Static Abstract Type Factory—can be applied additionally to build such Static Frameworks [Bac06].

There are many more patterns and idioms that assist in the development of concurrent applications. Schmidt et al. [SSRB02e] and Mattson et al. [MSM04] are comprehensive collections or pattern languages in this area.

Acknowledgements

Without the invaluable feedback of Berna L. Massingill, who was the PLoP shepherd of this work, this paper would not have been the way it is now.

Last but not least my thanks and love go to Cornelia Kneser, my wife, for her constant support throughout the writing of the paper.

This work was supported by the Institute for Medical Informatics and Biostatistics, Basel, Switzerland.

References


[Bac06] Bachmann, Philipp: Static and metaprogramming patterns to facilitate portable code. In Proceedings of the 13th Pattern Languages of


REFERENCES

England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, dritter, unveränderter Nachdruck Auflage, 1996. German translation of “Design Patterns. Elements of Reusable Object–Oriented Software”.


[Lak96] Lakos, John: Large–Scale C++ Software Design. Addison–Wesley Professional Computing Series; ed. by BRIAN W. KERNIGHAN. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney ·
References


23
REFERENCES


[Str94] Stroustrup, Bjarne: Design und Entwicklung von C++. Addison–Wesley, Bonn · Paris · Reading, Massachusetts · Menlo Park, California · New York · Don Mills, Ontario · Workingham, England ·
Amsterdam · Milan · Sydney · Tokyo Singapore · Madrid · San Juan · Seoul · Mexico City · Taipei, Taiwan, 1994, ISBN 3-89319-755-9. German translation of “The Design and Evolution of C++”.


