Investigating the Proactor design pattern

Martin Matusiak

November 29, 2007

Universiteit Utrecht

Outline

Universiteit Utrecht

Disclaimer

What to expect from this talk

- My impression: Proactor is a complicated pattern.
- I do not follow the textbook, instead I give you my thoughts/conclusions based on some extra background reading.
- Hoping to fill in the blanks you may have had while reading about it.
- Proactor seems very similar to Reactor, I go in depth on this.
- Questions underway? Feel free to interrupt.

Based on these sources

1. *Pattern-Oriented Software Architecture : Patterns for Concurrent and Networked Objects* [our textbook]
   2000, Schmidt et al.
2. *Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events*
   1997, Schmidt et al.
3. *Applying the Proactor Pattern to High-Performance Web Servers*
   1998, Schmidt et al.
4. *ProActor vs Reactor: Comparing Two High-Performance I/O Design Patterns*
   2005?, Libman & Gilbourd

Characteristics

- Quite old and possibly outdated sources.
- Seemingly little interest and/or knowledge of Proactor/Reactor patterns outside Schmidt and friends.

Proactor pattern in context

What is it good for?

- I/O! (no other use case given in sources)

Proactor pattern in context

What is it good for?

- I/O! (no other use case given in sources)
- *More specifically:* how to process I/O events from multiple sources concurrently.

What is it good for?

- I/O! (no other use case given in sources)
- *More specifically:* how to process I/O events from multiple sources concurrently.

- *(++More) specifically:* how to decouple I/O concurrency from process concurrency (n:m relation)

Outline

Universiteit Utrecht

Synchronous, blocking I/O

```
lines = read("info.txt");
//  thread blocks until I/O call completes
//  moved to I/O wait list, not runnable

//  * wait until call completes *

//  thread status changed back to runnable
print(lines);
```

Synchronous, non-blocking I/O

```
myhandle = from_filename("info.txt");

while (!lines) {
  result = select(myhandle, timeout);
  //  thread switched to I/O bound

  //  * wait until call completes *

  //  thread switched back to cpu bound

  if (result) {
    lines = read(handle);
    //  syscall has confirmed handle is ready, I/O call
    //  is synchronous and non-blocking
  }
}
print(lines);
```

Asynchronous I/O

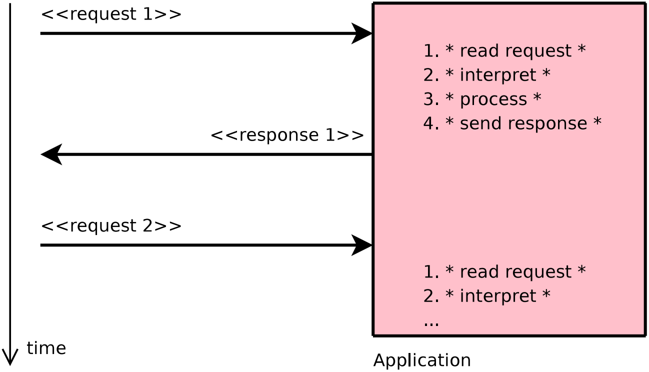
```
int main {
  read("info.txt");
  //  call is asynchronous, thread still cpu bound
  //  I/O operation executes in kernel thread,
  //   not application thread
}

void handle_read(string lines) {
  print(lines);
}
```

Missing some more OS-specific plumbing, but this is the basic idea.
Note: call site differs from return site.

Outline

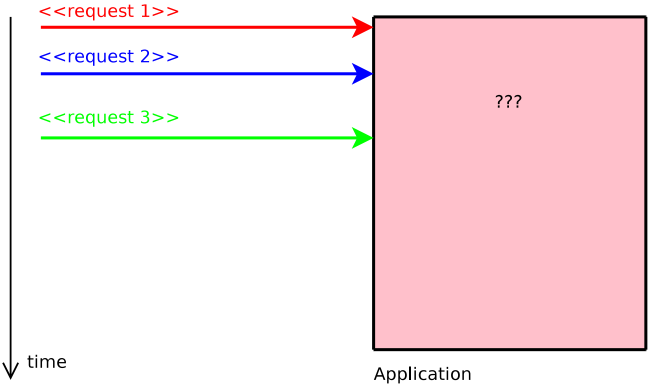
Handling I/O in a web server

Single source, sequential I/O, “**one at a time**”
$\longrightarrow$ single thread



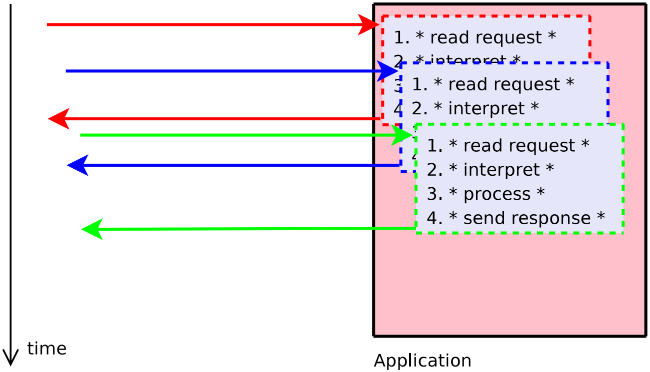
Handling I/O in a web server

Multiple source, concurrent I/O, “**many at a time**”



Handling I/O in a web server

Multiple source, concurrent I/O, “**many at a time**”
$\rightarrow$ thread per request, *multi-threaded synchronous I/O*

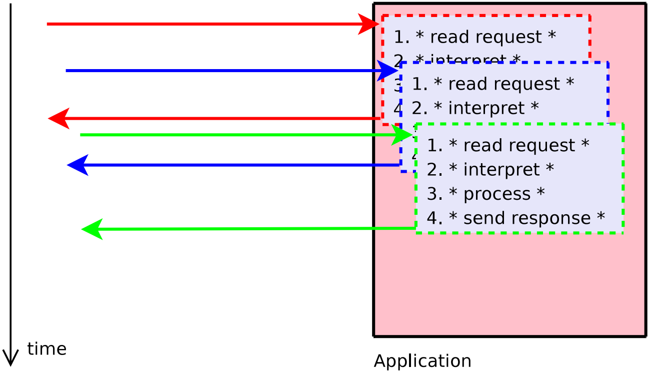


Handling I/O in a web server

Multiple source, concurrent I/O, **“many at a time”**

→ thread per request, *multi-threaded synchronous I/O*

✔ requests are independent (threads have no shared state)

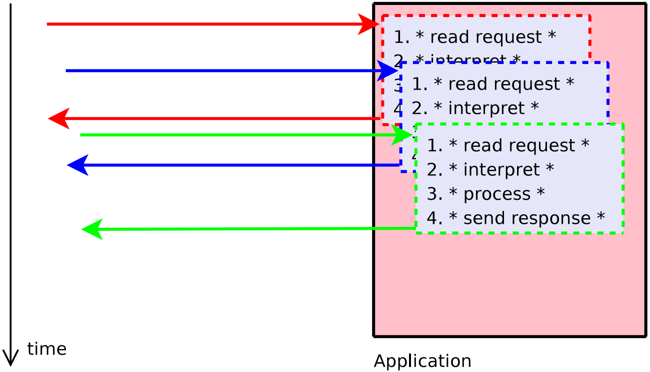


Handling I/O in a web server

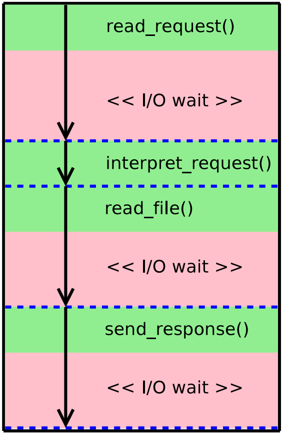
Multiple source, concurrent I/O, **“many at a time”**

→ thread per request, *multi-threaded synchronous I/O*

- ✔ requests are independent (threads have no shared state)
- ☠ context switching



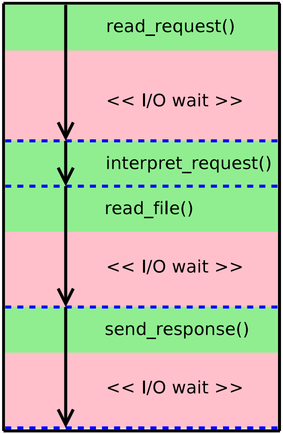
Multi-threaded synchronous I/O



The life of a thread

- thread lifetime = request processing duration

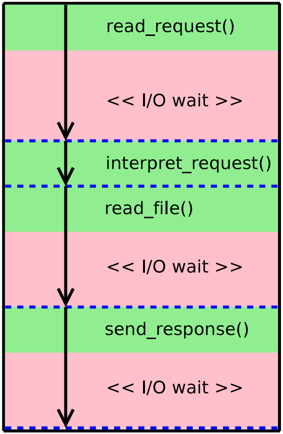
Multi-threaded synchronous I/O



The life of a thread

- thread lifetime = request processing duration
- spends most of the time waiting

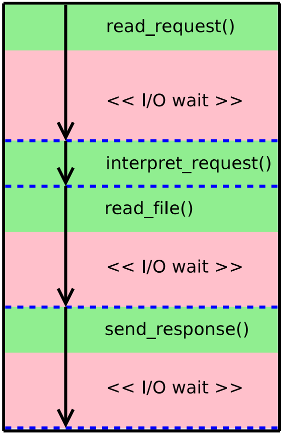
Multi-threaded synchronous I/O



The life of a thread

- thread lifetime = request processing duration
- spends most of the time waiting
- *has to be in memory*

Multi-threaded synchronous I/O

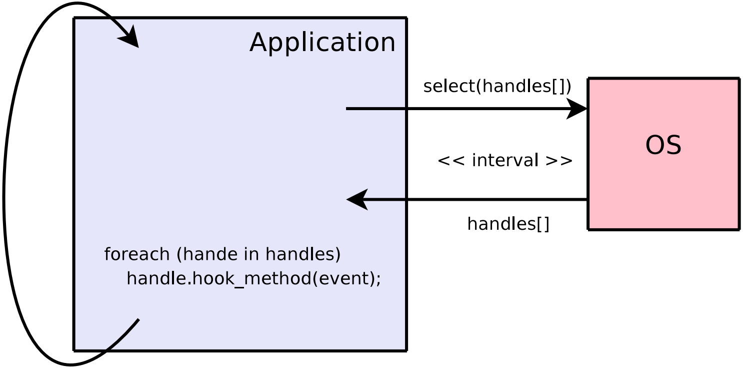


The life of a thread

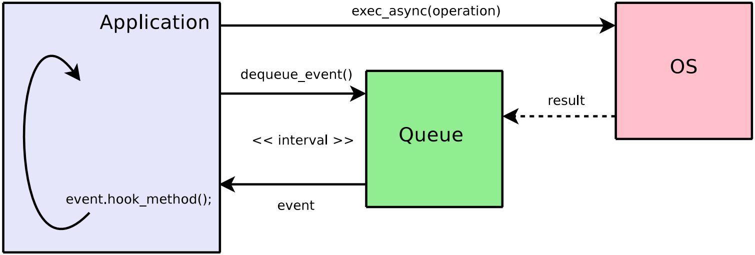
- thread lifetime = request processing duration
- spends most of the time waiting
- *has to be in memory*
- *has to be runnable (ie. subject to context switches)**

Outline

The Reactor pattern



The Proactor pattern

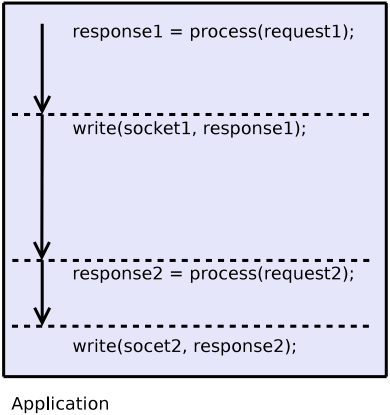


Reactor vs Proactor

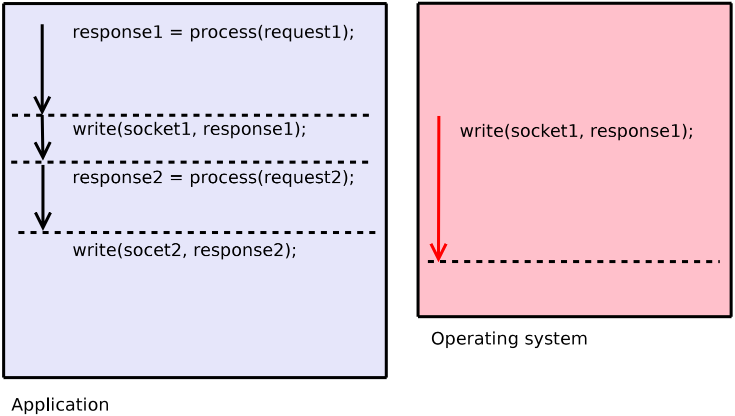
So what is the difference exactly?

- The patterns are very similar.
  - A single thread of execution.
  - Multiple I/O operations ongoing simultaneously.
  - Execution of event handlers is sequential.
- They differ in minor details.
  - Proactor *launches* (or even generates) I/O events and waits for callbacks, ie. **proactive stance**.
    Reactor only *anticipates* [external] events, ie. **reactive stance**.
  - Proactor delegates I/O operations to the OS – **asynchronous**
    Reactor runs I/O operations in its own thread – **synchronous**

Reactor: synchronous I/O



Proactor: asynchronous I/O



Comparison revisited

- Pattern wise they are almost the same.
  - You could even say they are compositions of each other. *TProactor (Terabit Solutions) is a proactor emulated on top of a reactor. (no performance degradation)*

Comparison revisited

- Pattern wise they are almost the same.
  - You could even say they are compositions of each other. *TProactor (Terabit Solutions) is a proactor emulated on top of a reactor. (no performance degradation)*
- The difference is in how they use OS services for I/O.
  - The distinction is therefore in use of OS, not in design.
  - Proactor is just a way of taking advantage of async I/O support in the OS.

Comparison revisited

- Pattern wise they are almost the same.
  - You could even say they are compositions of each other. *TProactor (Terabit Solutions) is a proactor emulated on top of a reactor. (no performance degradation)*
- The difference is in how they use OS services for I/O.
  - The distinction is therefore in use of OS, not in design.
  - Proactor is just a way of taking advantage of async I/O support in the OS.
- Is asynchronous I/O faster than synchronous I/O? If so, why?
  - Kernel design territory, not application/middleware design.

Outline

A web server based benchmark

- Source: *Applying the Proactor Pattern to High-Performance Web Servers*
- Experiment setup:
  - *JAWS* web server with pluggable concurrency strategies:
    - thread per request
    - thread pool
    - proactor pattern
  - Method: generate `GET` requests for documents of a given size, steadily increase number of concurrent clients.
  - Measurements:
    - Throughtput (data per unit of time)
    - Latency (time elapsed from request to response)

Benchmark: 5kb files

Throughput: higher is better

Latency: lower is better

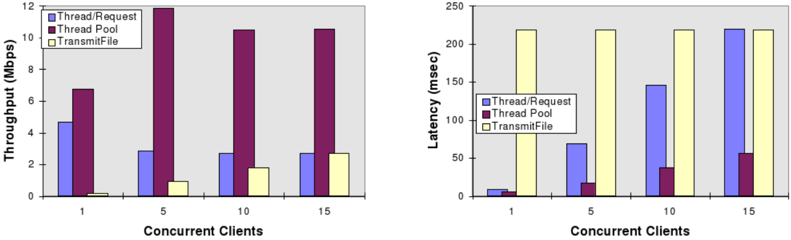


Figure 7: Experiment Results from 5K File

Most common size of documents on a typical web server.

Benchmark: 50kb files

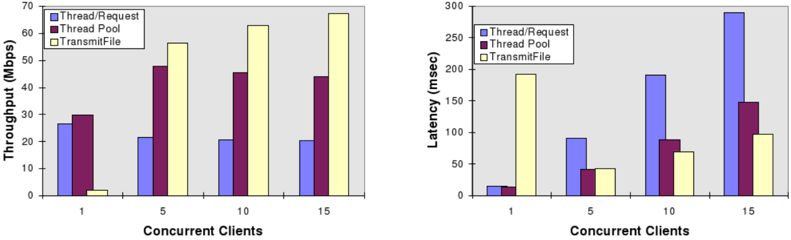


Figure 8: Experiment Results from 50K File

Second most common size of documents on a typical web server.

Universiteit Utrecht

Benchmark: 500kb files

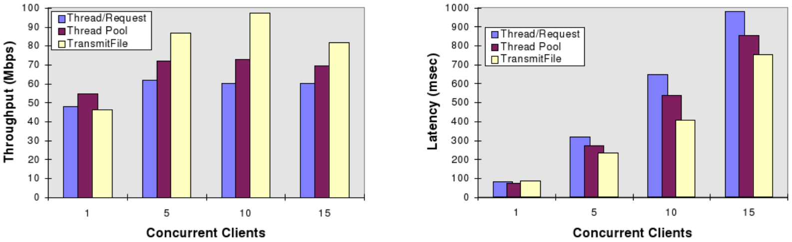


Figure 9: Experiment Results from 500K File

Break point reached, proactor based profile (TransmitFile) has highest throughput and lowest latency.

Benchmark: 5mb files

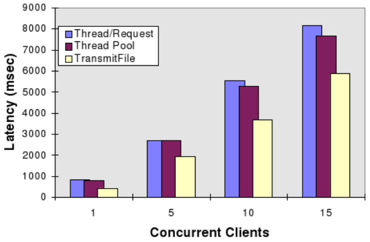


Figure 10: Experiment Results from 5M File

Outline

What it's all about

I/O strategies

The case for Proactor

But didn't we have Reactor for that?

But is it any good?

Proactors in the wild?

Operating System support for I/O

Synchronous non-blocking I/O

- `select` **nix, BSD, Windows*
- `poll` **nix, BSD*
- `epoll` (maintains state, more efficient) *Linux*
- `KQueue` (similar to epoll) *BSD*
- `/dev/poll` *Solaris*
- `libevent` (encapsulates OS-specific interfaces) *Linux, BSD, Solaris, Windows*

Operating System support for I/O

Synchronous non-blocking I/O

- `select` **nix, BSD, Windows*
- `poll` **nix, BSD*
- `epoll` (maintains state, more efficient) *Linux*
- `KQueue` (similar to epoll) *BSD*
- `/dev/poll` *Solaris*
- `libevent` (encapsulates OS-specific interfaces) *Linux, BSD, Solaris, Windows*

Asynchronous I/O

- Completion ports, overlapped I/O *Windows NT* (current?)
- POSIX AIO *Linux, BSD, others?*
- `io_submit()`, `io_getevents()` *Linux*

Real world applications

Twisted framework for Python

- *“`Twisted` is an event-driven networking engine written in Python“*
- Borrows heavily from the ACE framework (Schmidt et al.)

Real world applications

Twisted framework for Python

- *"`Twisted` is an event-driven networking engine written in Python"*
- Borrows heavily from the ACE framework (Schmidt et al.)
- *Reactor* pattern used as one of the basic building blocks.

Twisted framework for Python

- *"`Twisted` is an event-driven networking engine written in Python"*
- Borrows heavily from the ACE framework (Schmidt et al.)
- *Reactor* pattern used as one of the basic building blocks.
- Originally created as a reusable network library for game development.

Real world applications

Twisted framework for Python

- *"`Twisted` is an event-driven networking engine written in Python"*
- Borrows heavily from the ACE framework (Schmidt et al.)
- *Reactor* pattern used as one of the basic building blocks.
- Originally created as a reusable network library for game development.
- *Twisted Core* is an async event loop geared at fast development of various async client and server applications.

Real world applications

Twisted framework for Python

- *“`Twisted` is an event-driven networking engine written in Python“*
- Borrows heavily from the ACE framework (Schmidt et al.)
- *Reactor* pattern used as one of the basic building blocks.
- Originally created as a reusable network library for game development.
- *Twisted Core* is an async event loop geared at fast development of various async client and server applications.

No uses of *Proactor* pattern found. ☹

Discussion topics

- Is the pattern clear to you? Is it straightforward or is it subtle?
- Does the complexity of the Proactor pattern intimidate you?
- Do you believe Reactor and Proactor differ significantly beyond the evidence presented?