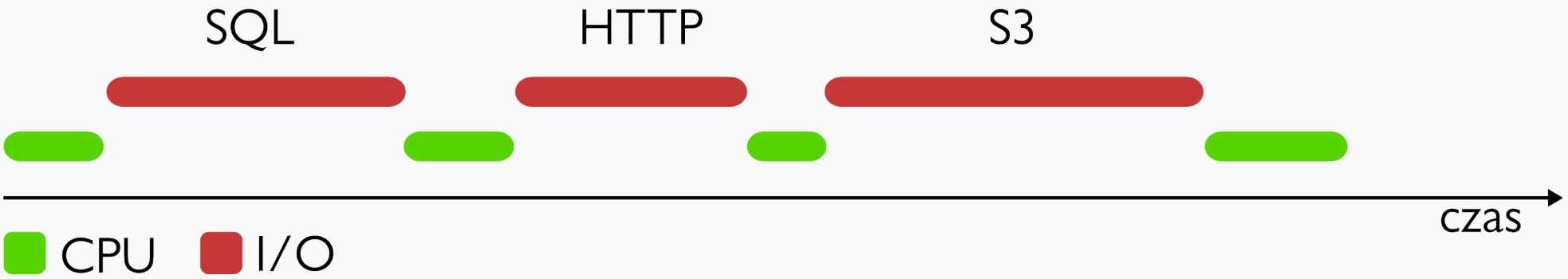
EventMachine. Asynchroniczne I/O w Ruby

Paweł Pacana <pawel.pacana@gmail.com>

Efektywne wykorzystanie maszyny

- maksymalizacja wykorzystania zasobów
- ~80% (Google)

CPU vs. I/O



Blokujące I/O

- pliki
- baza danych
- memcache
- rozwiązywanie nazw DNS
- system()

Problem **C10K**

- http://www.kegel.com/c10k.html
- 10000 równoczesnych połączeń

Rozwiązania (I)

- **blokujące, synchroniczne I/O**
  - wątki
    - Ruby
      - 1.8 i GreenThreads
      - 1.9 i GIL
  - procesy (fork)

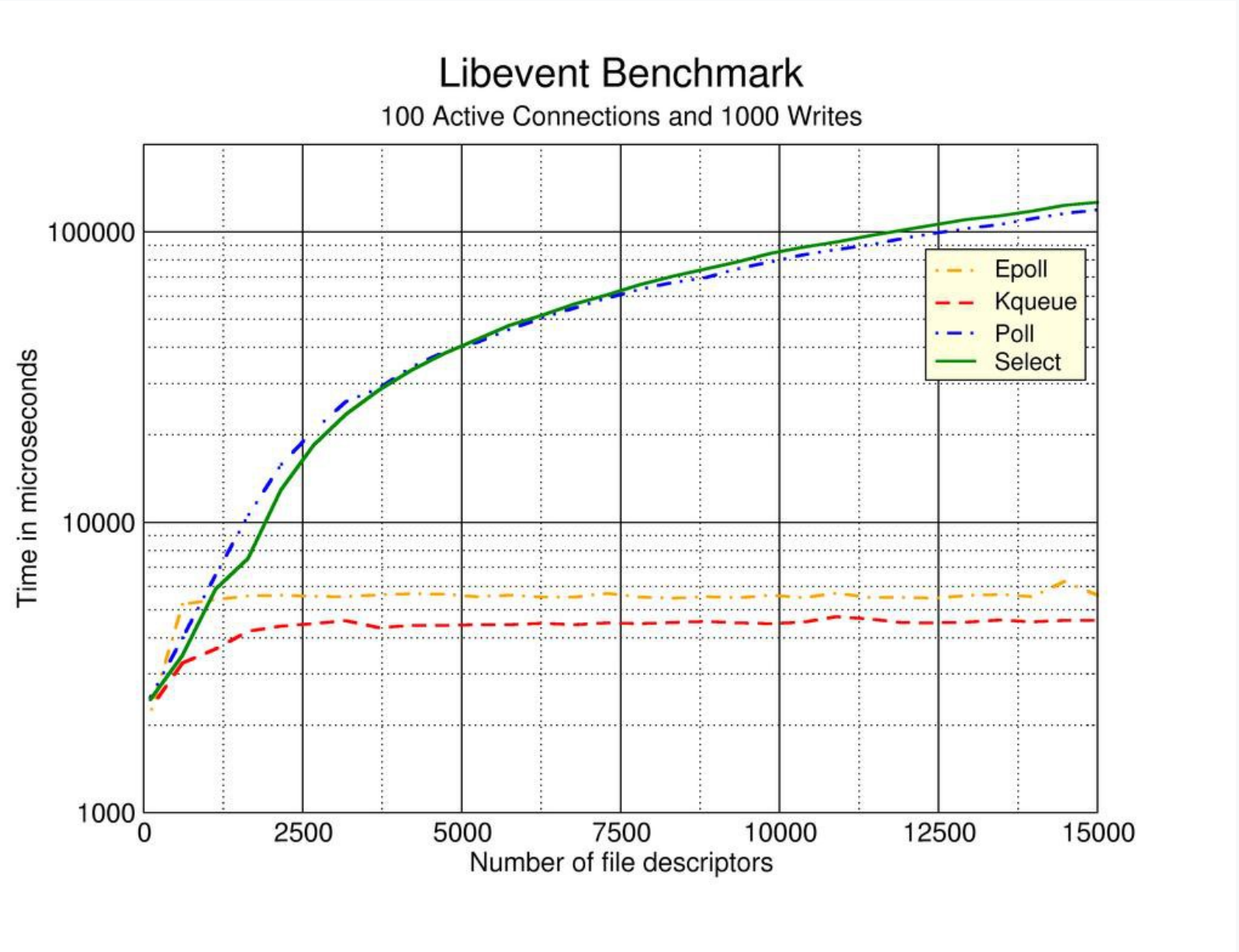
Rozwiązania (2)

- **nieblokujące I/O + polling**
  - select()
    - nieliniowa wydajność
    - limit FD_SETSIZE
  - poll()
    - bez limitu
    - wolny przy kilku tyś. deskryptorów (przeszukiwanie)

Rozwiązania (3)

- **asynchroniczne I/O**
  - kqueue (FreeBSD), epoll (Linux 2.6)
  - kolejkowane sygnały po wykonaniu operacji

(2) vs. (3)


Libevent Benchmark
100 Active Connections and 1000 Writes
Time in microseconds
100000
10000
1000
0
2500
5000
7500
10000
12500
15000
Number of file descriptors
Epoll
Kqueue
Poll
Select

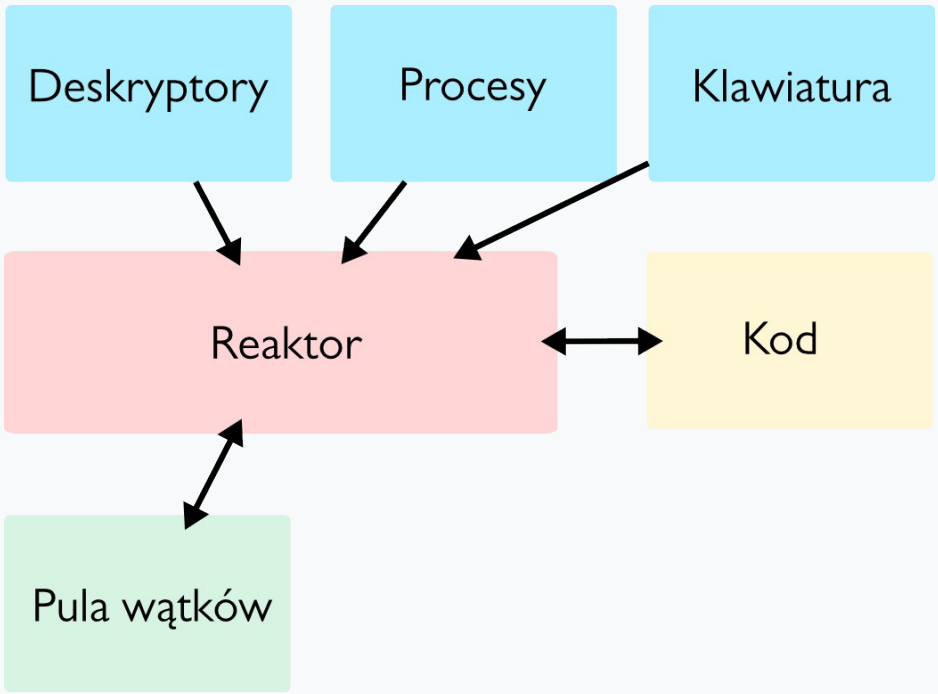
Reactor pattern

*„Concurrent programming pattern for handling service **requests** delivered **concurrently** to a service handler by **one or more inputs**. The service handler then **demultiplexes** the incoming requests and dispatches them **synchronously** to the associated request handlers."*

- EventMachine, Rev (Ruby)
- Twisted, Tornado, asyncore (Python)
- nodejs (Javascript)
- Netty (Java)

EventMachine

- jednowątkowy
- jednoczesne przeplatanie wieli operacji I/O i jednej operacji CPU
- implementacja protokołów sieciowych
- pula wątków



EchoServer (I)

```
require 'eventmachine'

module EchoServer
  def receive_data(data)
   send_data(data)
  end
end

EM.run do
  EM.start_server "0.0.0.0", 10000, EchoServer
end
```

EchoServer (2)

```
require 'eventmachine'
class EchoServer < EM::Connection
  attr_accessor :config

  def receive_data(data)
   send_data(data)
  end
end

EM.run do
  EM.start_server "0.0.0.0", 10000, EchoServer do |server|
    server.config = ...
  end
end
```

Timer

```
require 'eventmachine'

EM.run do
 EM.add_timer(5) do
   puts "BOOM"
   EM.stop
 end
 EM.add_periodic_timer(1) do
   puts "Tick ... "
 end
end
```

Delay

```
require 'eventmachine'

EM.run do
 EM.add_timer(5) do
   EM.next_tick do
    EM.stop
   end
 end
end
```

Defer

```
require 'eventmachine'
require 'thread'

EM.run do
 EM.add_timer(2) do
   puts "Main #{Thread.current}"
   EM.stop
 End

 EM.defer(proc { puts "Defer #{Thread.current}" })
end
```

Deferrable

```
class MyDeferrable
  include EM::Deferrable
  def go(str)
    puts "Go #{str} go"
  end
end

EM.run do
  df = MyDeferrable.new

  df.callback do |x|
    df.go(x)
    EM.stop
  end

  EM.add_timer(1) do
    df.set_deferred_status :succeeded, "SpeedRacer"
  end
end
```

Spawning

```
require 'rubygems'
require 'eventmachine'

EM.run do
 s = EM.spawn do |val|
   puts "Received #{val}"
 End

 EM.add_timer(1) do
   s.notify "hello"
 End

 EM.add_timer(3) do
   EM.stop
 end
end
```

Klient sieciowy

```
require 'eventmachine'

class Connector < EM::Connection
 def post_init
   puts "Getting /"
   send_data "GET / HTTP/1.1\r\nHost: MagicBob\r\n\r\n"
 end

 def receive_data(data)
   puts "Received #{data.length} bytes"
 end
end

EM.run do
 EM.connect "www.google.com", 80, Connector
end
```

Protokoły

- http://wiki.github.com/eventmachine/eventmachine/protocol-implementations
- wbudowane:
  - HTTP
  - SMTP
  - STOMP
  - SASLAuth
  - Postgres
  - LineText
  - ...
- zewnętrzne:
  - IRC
  - DNS
  - MongoDB
  - XMPP
  - SNMP
  - ...

```
class KeyboardHandler <
EM::Connection
  include EM::Protocols::LineText2
  def receive_line(line)
    case(line)
    when /^exit$/ then
      EM.stop
    end
  end
end
EM::run do
  EM.open_keyboard(KeyboardHandler)
end
```

Aplikacje na EventMachine

- Thin
- EventedMongrel
- Cramp
- GitHub: 66 projektów

Q&A

- http://rubyeventmachine.com/