Du bon usage des continuations pour la programmation de serveurs pour le ouèbe

Christian Queinnec UPMC–LIP6

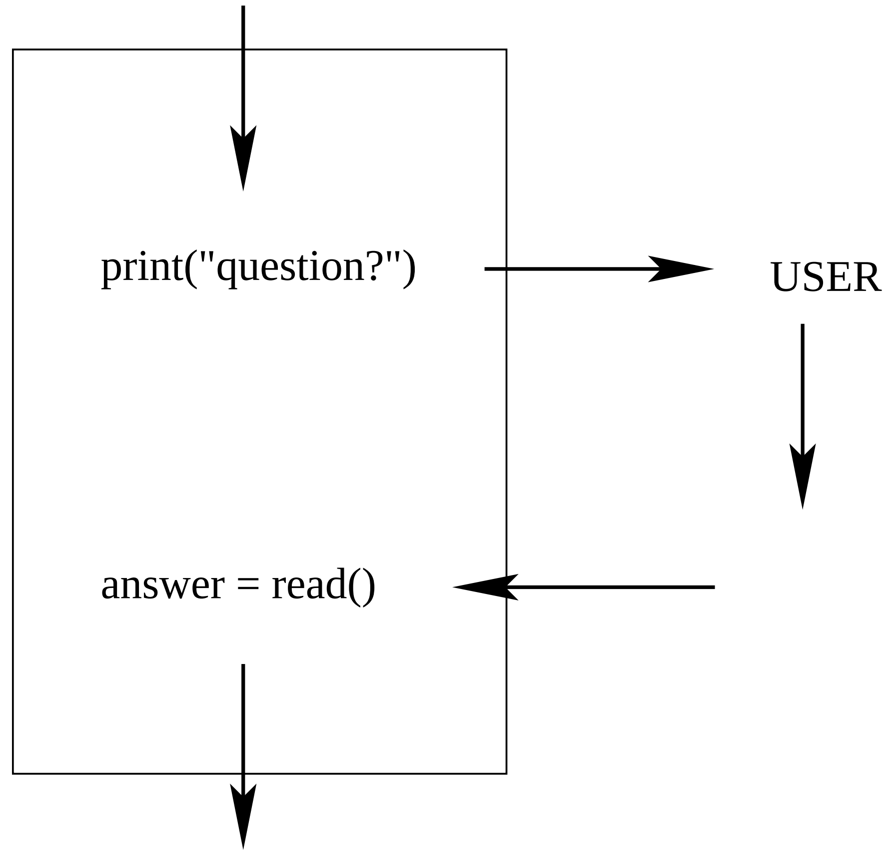
Bits of history

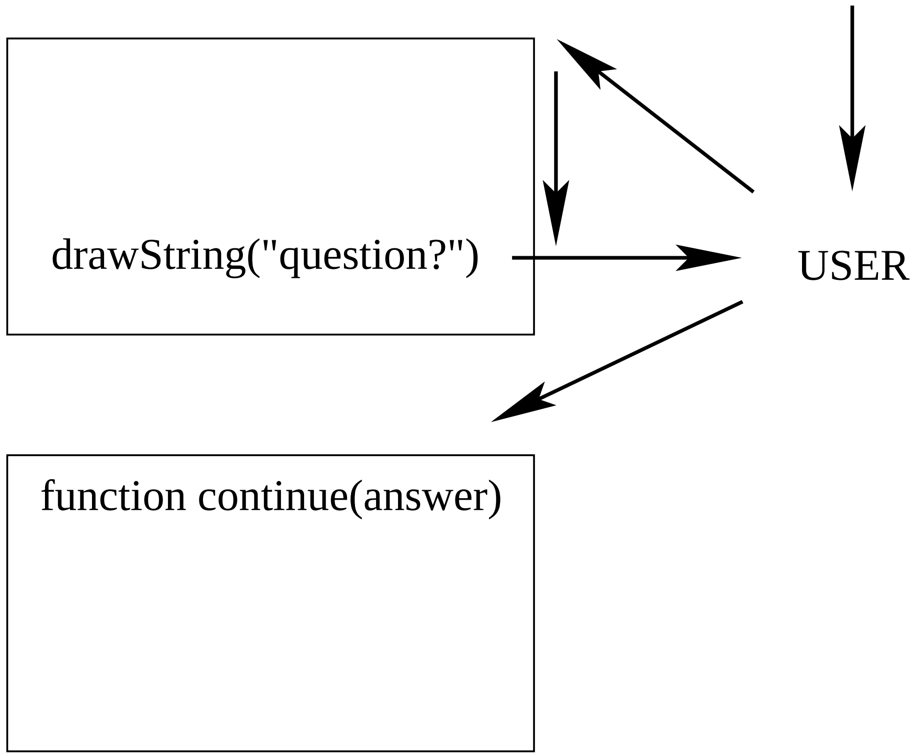
The single important thing:

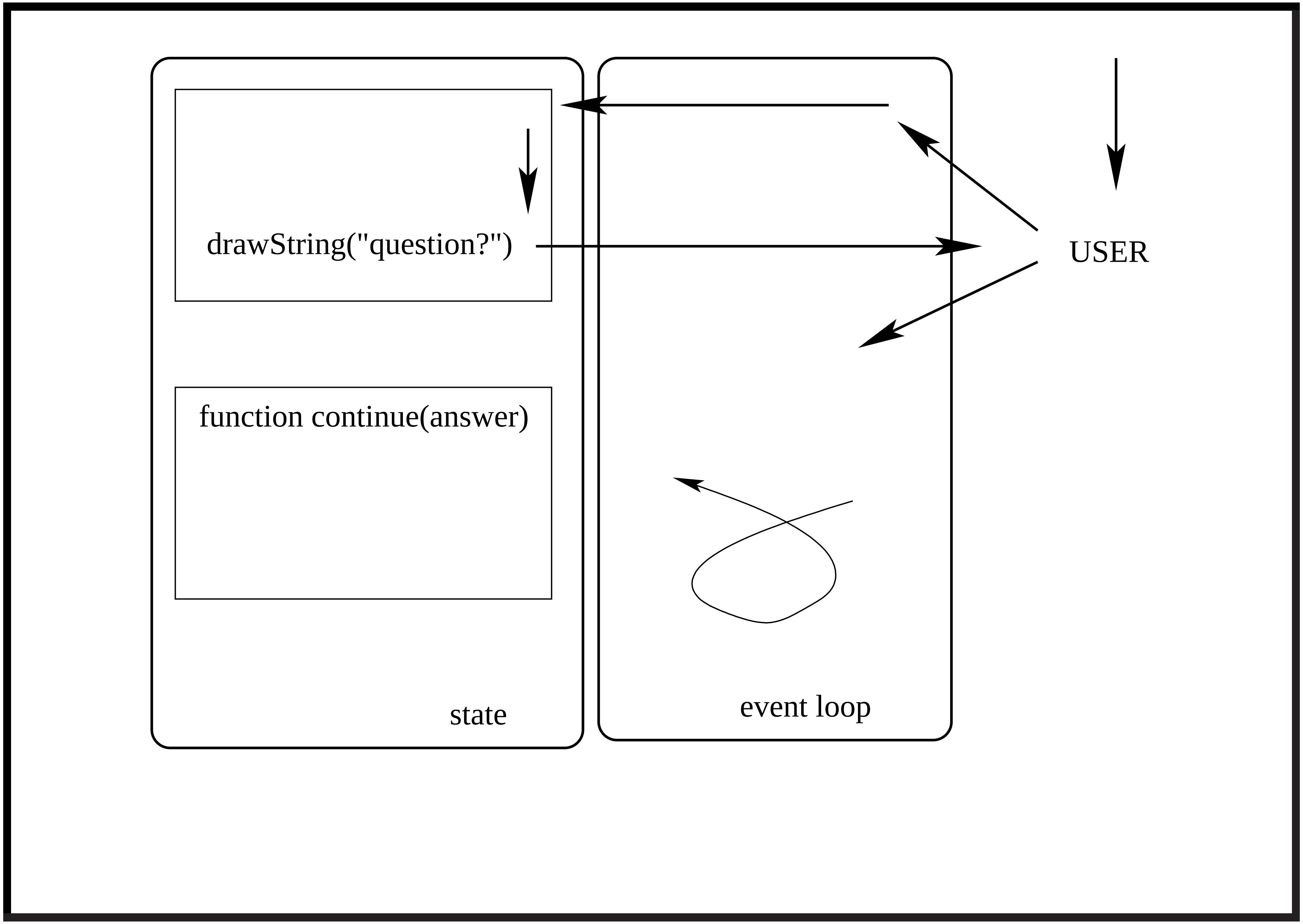
Bits of history

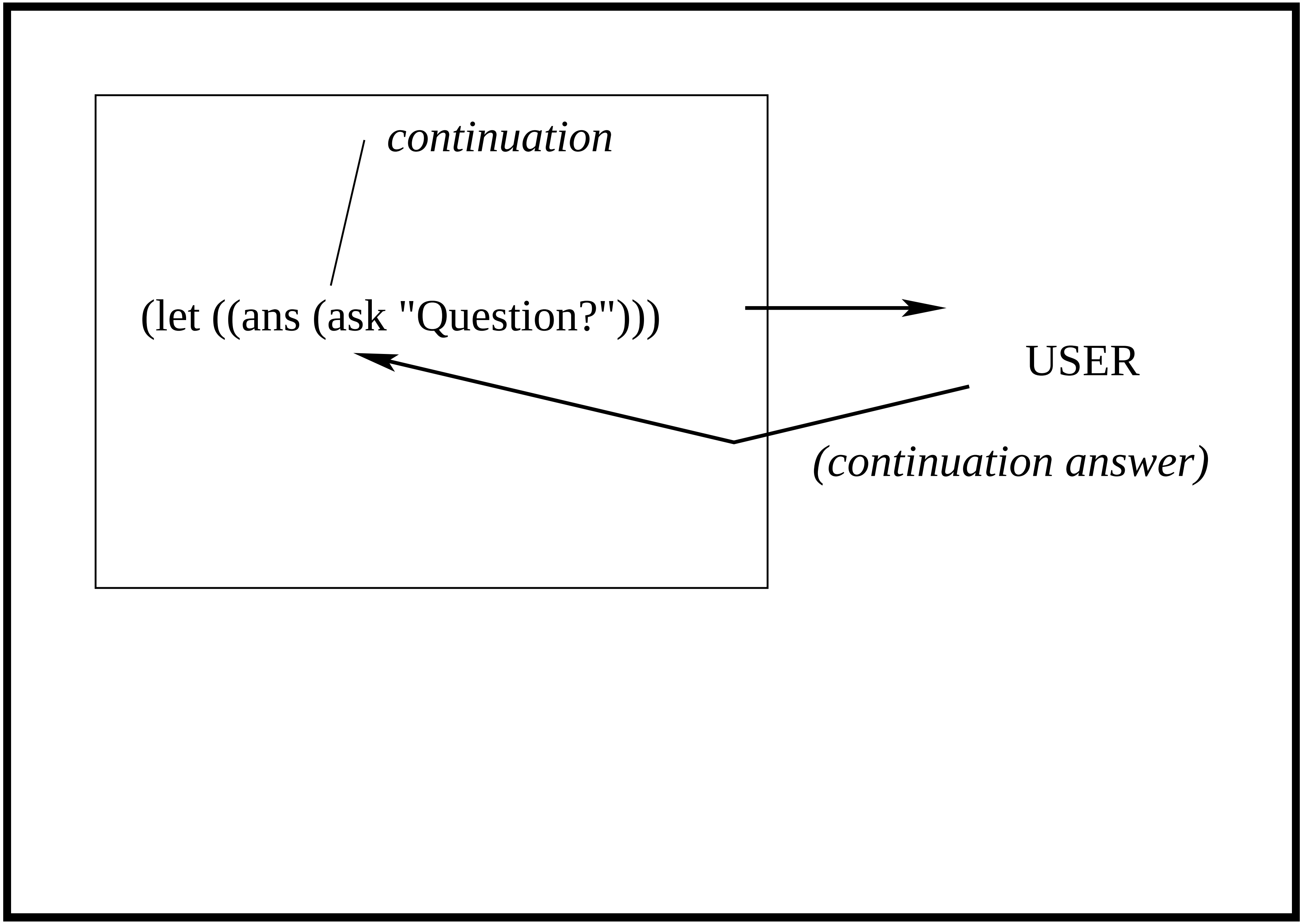
The single important thing:

code!


print("question?")
USER
answer = read()


drawString("question?")
USER
function continue(answer)


drawString("question?")
function continue(answer)
state
event loop
USER


continuation
(let ((ans (ask "Question?")))
USER
(continuation answer)

A Web computation

Here starts the demonstration.

The URL looks like:

```
http://127.0.0.1:19891/          ; On my machine,
 ps3i                            ; a Scheme interpreter,
     ? action=eval               ; has to evaluate
       expression=               ; the content of a file:
 (load "http://127.0.0.1:19891/icfp/icfp2000b.scm")
```

Read the source!

```
(define (multiplication-demo)
  (let ((n1 (read-first-number)))
    (let ((n2 (read-second-number n1)))
      (show (mk-result-page n1 n2)) ) ) )

;;;Go!
(multiplication-demo)
```

```
(define (mk-result-page n1 n2)
  (make-page
   (lambda (kUrl)
     (html (head (title "Result") (common-css))
           (body (generate-multiplication
                  n1 n2 (* n1 n2) )
                 (buttons 'back) ) ) ) ) )
```

```
(define (mk-result-page n1 n2)
  (make-page
   (lambda (kUrl)
     (html (head (title "Result") (common-css))
           (body (generate-multiplication
                  n1 n2 (* n1 n2) )
                 (buttons 'back) ) ) ) ) )
```

The LAML experience (Kurt Nørmark), or XML re-parenthetized as Sexpressions!

```
(define (generate-multiplication n1 n2 product)
  (table (tr (td align: 'right       n1    ))
         (tr (td align: 'right " * " n2    ))
         (tr (td align: 'right "-------"   ))
         (tr (td align: 'right   product   )) ) )
```

```
(define (read-first-number)
  (define (create-page)
    (make-page
     (lambda (kUrl)
       (html (head (title "First number?") (common-css))
             (body
              (quizz kUrl
                     "Give me a number, please? "
                     (quizz-text "n1" 1 5 "543")
                     (quizz-submit "Send!") )
              (buttons 'clone) ) ) ) ) )
  (let ((request (show (create-page))))
    (string->number
     (request-parameter-get request "n1") ) ) )
```

```
(define (read-second-number n1)
  (define (create-page n1)
    (make-page
     (lambda (kUrl)
       (html (head (title "Second number?") (common-css))
             (body
              (generate-multiplication n1 "..." "...")
              (quizz kUrl
                     "Give me a second number, please? "
                     (quizz-text "n2" 1 5 "297")
                     (quizz-submit "Send!") )
              (buttons 'clone 'back) ) ) ) ) )
  (let ((request (show (create-page n1))))
    (string->number
     (request-parameter-get request "n2") ) ) )
```

From the server-side

- How to `show` a page to a user ?
- How to handle a user's request ?

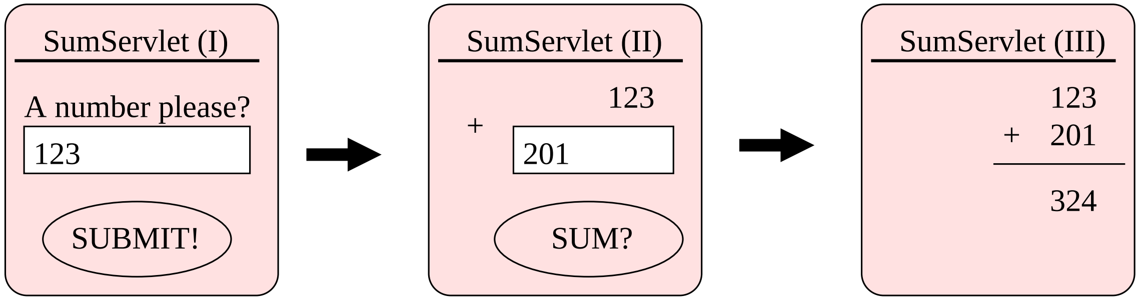
The `show` primitive

```
(define (show page)
  (call/cc
   (lambda (resume)
     (let ((url        (register-continuation! resume))
           (connection (current-connection)) )
       (display (page url) connection)
       (close-output-port connection)
       (suicide) ) ) ) )
```

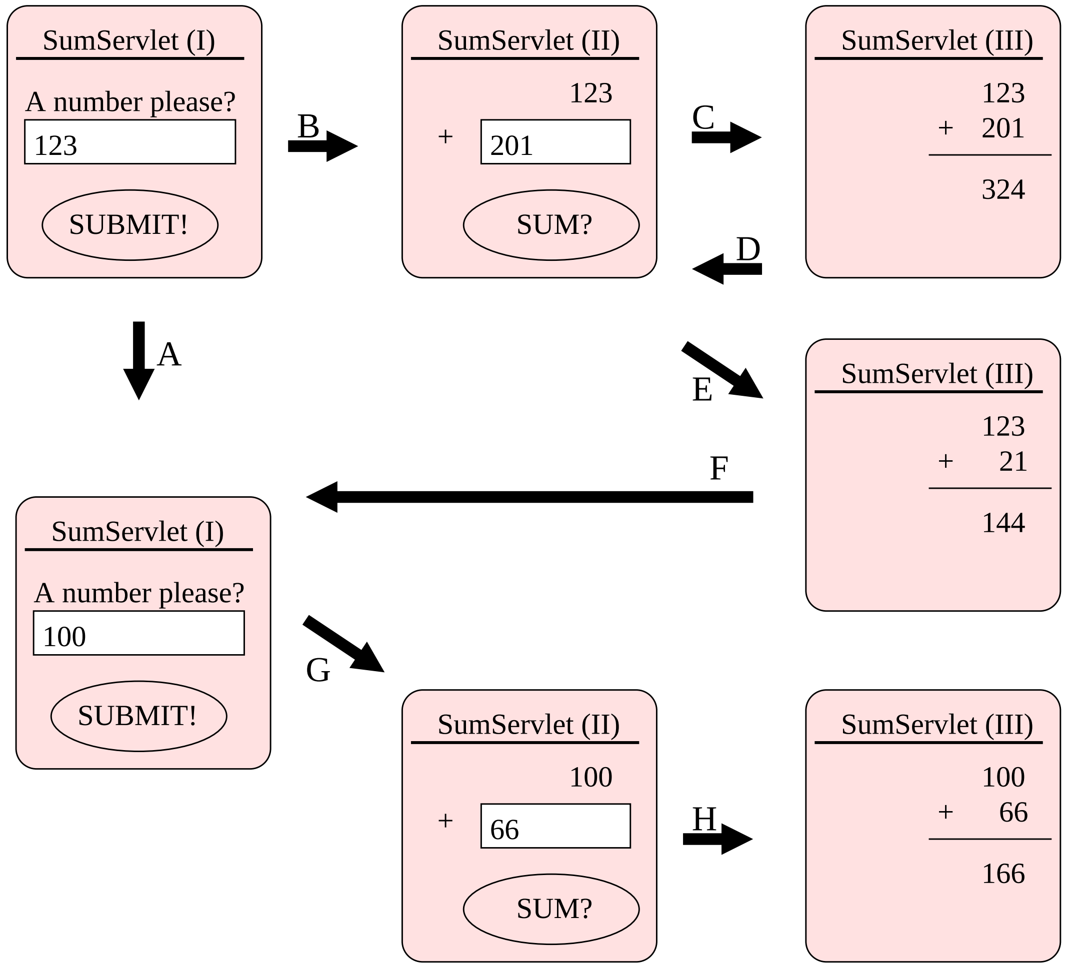
Server dispatcher

```
(define (server request)
  (let ((c (get-request-parameter request "continuation")))
    (let ((k (get-registered-continuation c)))
      (if k
          (k request)
          (error 'lost-continuation) ) ) ) )
```

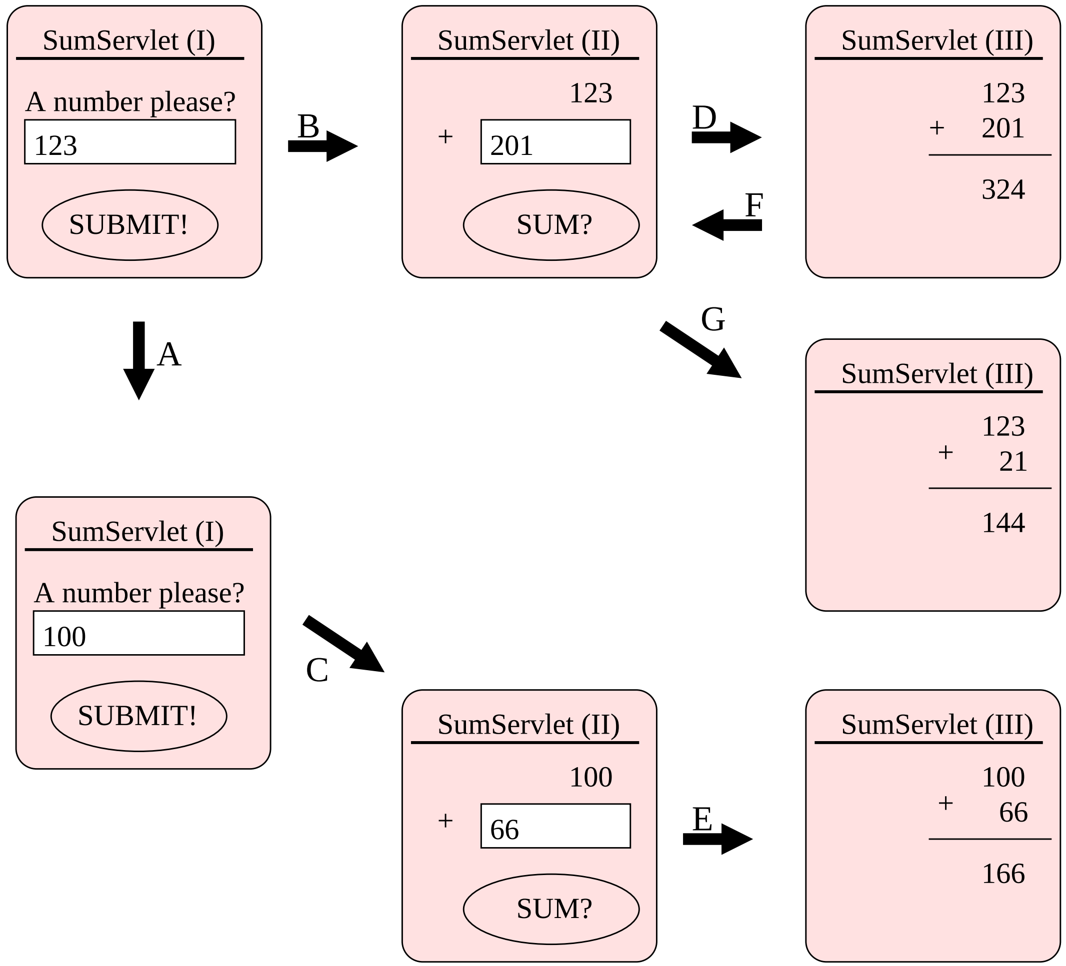
Interaction scenarii



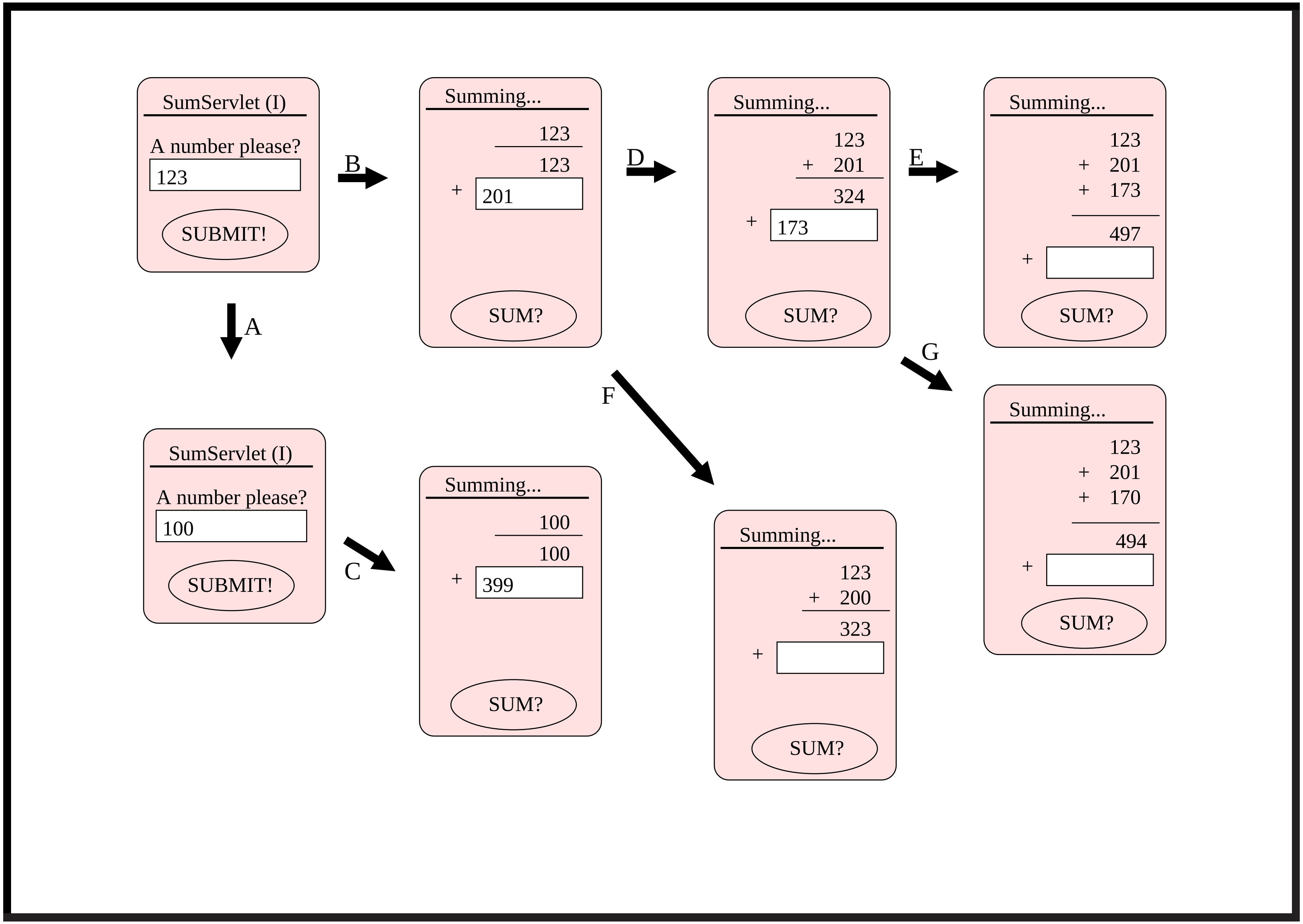

```
(let ((n1 (read-first-number)))
  (let ((n2 (read-second-number)))
    (display-result-page n1 n2) ) )
```



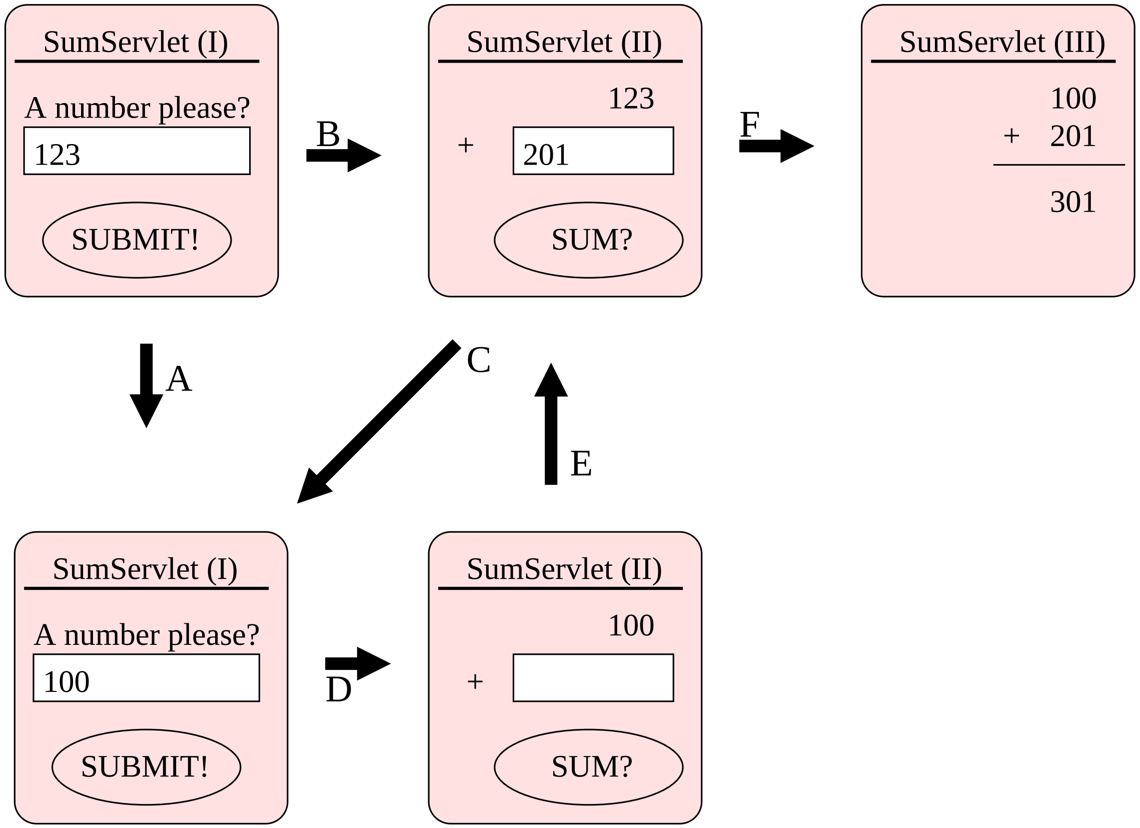
Sequential interaction.



Non sequential (concurrent) interaction.


SumServlet (I)
A number please?
123
SUBMIT!
A
SumServlet (I)
A number please?
100
SUBMIT!
B
C
Summing...
123
123
+ 201
SUM?
Summing...
100
100
+ 399
SUM?
D
F
Summing...
123
+ 201
324
+ 173
SUM?
Summing...
123
+ 200
323
+
SUM?
E
G
Summing...
123
+ 201
+ 173
497
+
SUM?
Summing...
123
+ 201
+ 170
494
+
SUM?

```
(define n1 0)                              ; global!
(set! n1 (read-first-number))
(display-result-page n1 (read-second-number n1))
```



Replay ?

```
(define n1 0)
(define n2 #f)      ;initialized to the boolean false
(set! n1 (read-first-number))
(if (not (integer? n2))
    (set! n2 (read-second-number)) )
(display-result-page n1 n2)
```

Never regress ?

```
(define there #f)
(define (progress f . args)
  (call/cc (lambda (k)
             (set! there k)
             (there (apply f args)) )) )
(define n1 0)
(set! n1 (progress read-first-number))
(set! n2 (progress read-second-number n1))
(progress (lambda () 'nothing))
(display-result-page n1 n2)
```

Problems

Theoretical interlude I

Concurrency primitives:

$$(\texttt{fork}\ \pi\ \pi')\big|_k \quad \equiv \quad \pi\ \big|_k \qquad || \qquad \pi'\ \big|_k$$

`(suicide)`

Browser induced concurrency

Instead of writing (`fork` π π'), just write:

```
(let ((request (show fork-page)))
  (let ((bool (get-request-parameter request "expression")))
    (if (equal? "true" bool) π π') ) )
```

Browser induced concurrency

Instead of writing `(fork` π π'`)`, just write:

```
(let ((request (show fork-page)))
  (let ((bool (get-request-parameter request "expression")))
    (if (equal? "true" bool) π π′) ) )
```

where `fork-page` contains
(where `kUrl` is the URL of the continuation):

```
window.open(kUrl + "?expression=true","uselessWindow","");
window.location.href = kUrl + "?expression=false";
```

Browser induced concurrency

Instead of writing (`fork` π π'), just write:

```
(let ((request (show fork-page)))
  (let ((bool (get-request-parameter request "expression")))
    (if (equal? "true" bool) π π′) ) )
```

where `fork-page` contains
(where `kUrl` is the URL of the continuation):

```
window.open(kUrl + "?expression=true","uselessWindow","");
window.location.href = kUrl + "?expression=false";
```

Milder analogy between `(suicide)` and closing a window.

Child-less

Demonstration

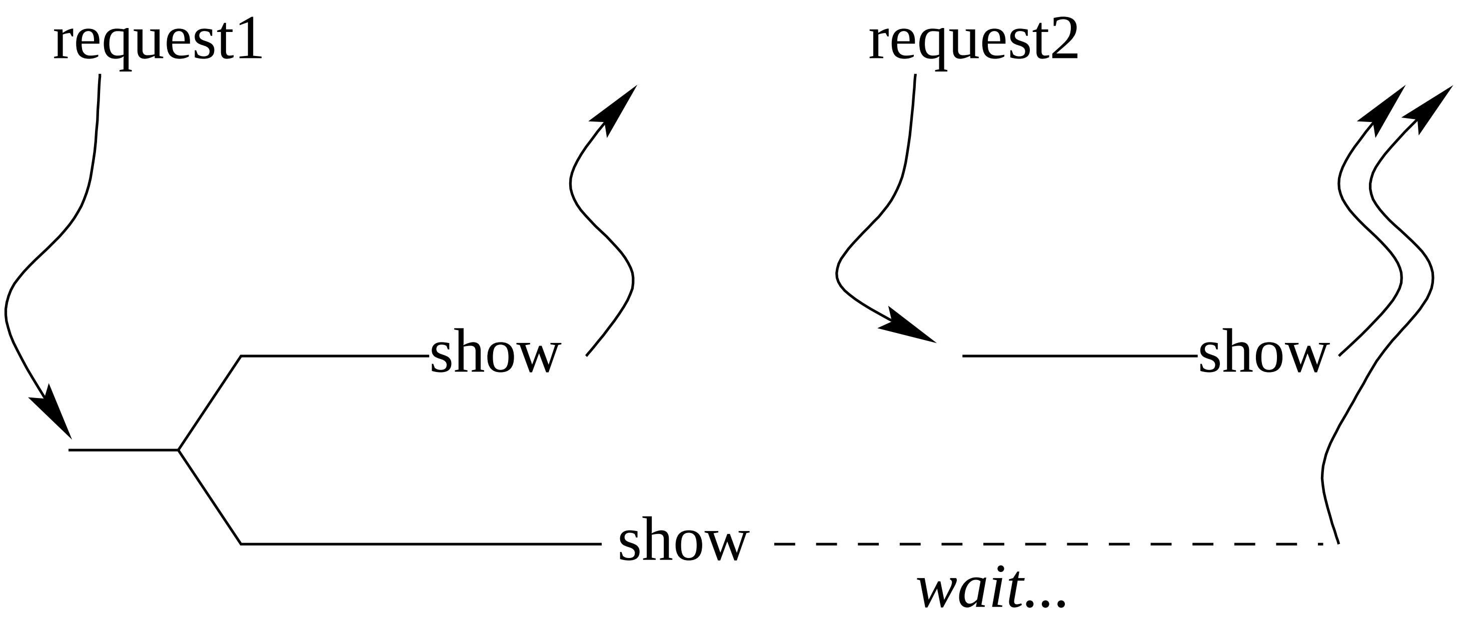
```
(fork (suicide)
      (begin (sleep 10) (suicide)) )
```

Orphan

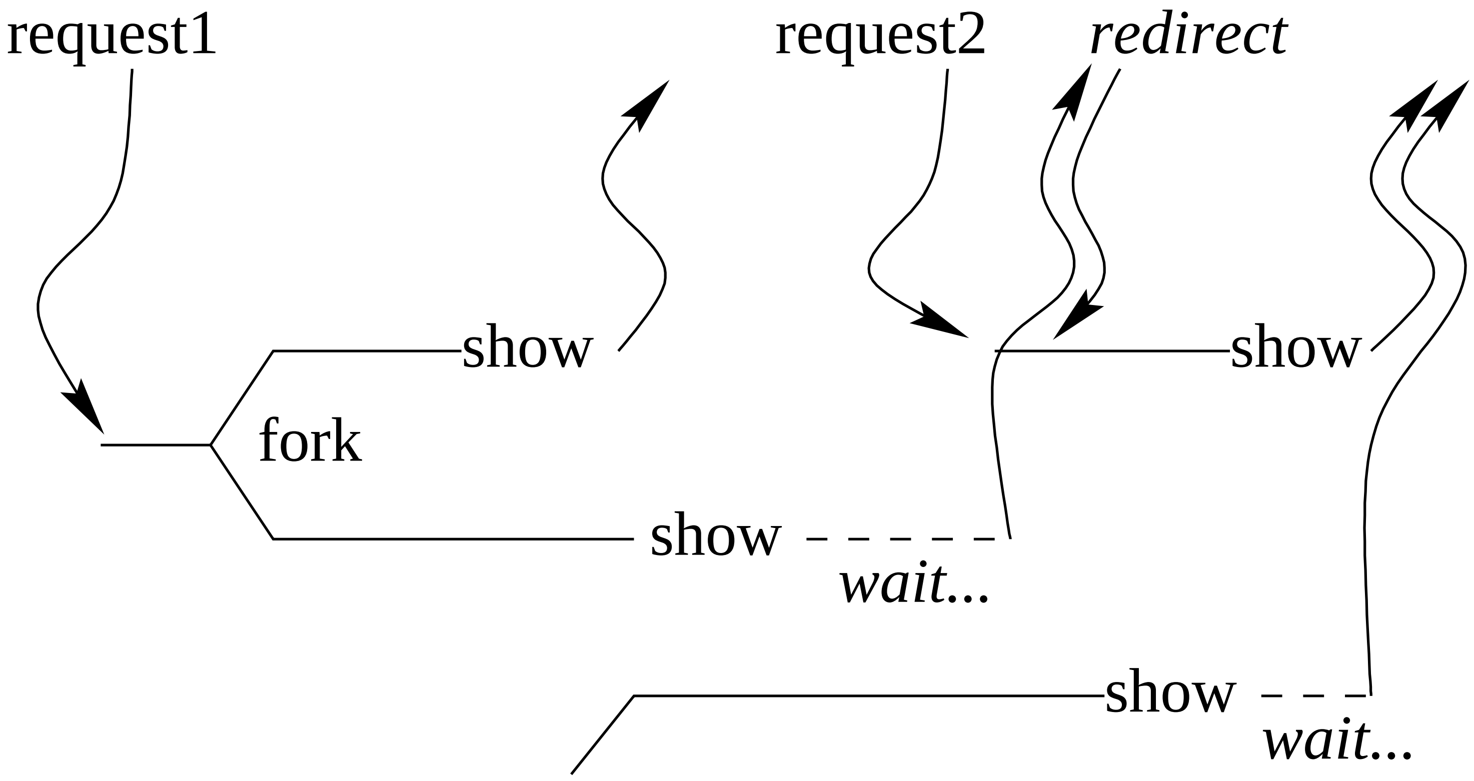
Demonstration

```
(fork (show (page 1))
      (begin (sleep 10) (show (page 2))) )
```

Use HTTP redirection:



Or even better:



Thread+offspring scope

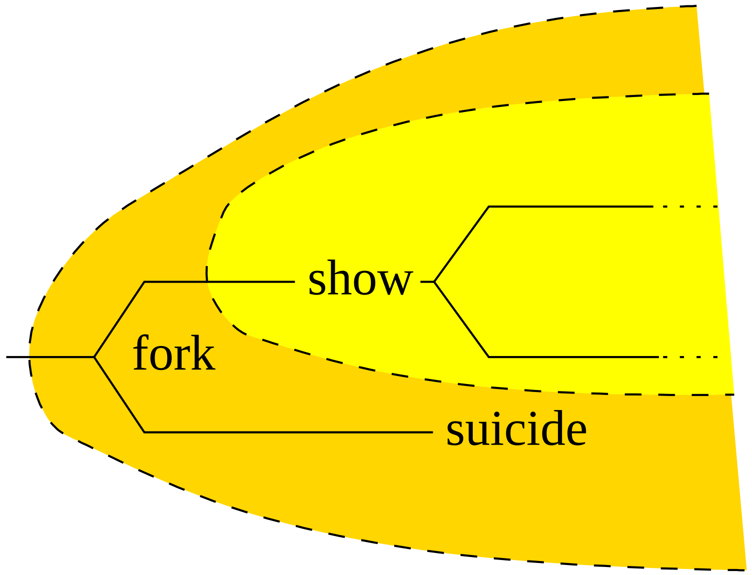
How to get the current connection as in the `show` function ? Invent the *Thread+offspring scope*!

A more colorful demonstration

Thread+offspring scope

How to get the current connection as in the `show` function ? Invent the *Thread+offspring scope*!

A more colorful demonstration



Two primitives:

(`thread+offspring-put` `'`*name* `'`*value*)

(`thread+offspring-get` `'`*name* `'`*value-if-absent*)

```
(define (read-first-number)
  (define (create-page)
    (make-page
     (lambda (kUrl)
       (html (head (title "First number?") (common-css))
             (body
              (quizz kUrl
                     "Give me a number, please?"
                     (quizz-text "n1" 1 5 "543")
                     (quizz-submit "Send!") )
              (buttons 'clone) ) ) ) ) )
  (let ((request (show (create-page))))
    (thread+offspring-set! 'bgcolor (choose-random-bgcolor))
    (string->number
     (request-parameter-get request "n1") ) ) )
```

Theoretical interlude II

In CPS++ style:

(`thread+offspring-put '`*name* `'`*value*`)`$|_k$ $\equiv$
`(dynamic-let ((`*name* `'`*value*`))`
`  (`*k* `#unspecified) )`

(`thread+offspring-get '`*name* `'`*value-if-absent*`)` $\equiv$
`(dynamic-ref `*name*`)`

Trails

A trail is a program that computes pages to serve.

It may include regular pages or, quizzes.

As well as some higher-level continuation operators.

Conclusions

- Direct style for web computations.

Conclusions

- Direct style for web computations.
- Continuations are useful.

Conclusions

- Direct style for web computations.
- Continuations are useful.

C'est fini!