

Implementierung eines Thread-Systems

Thread-Systeme

Ziel: Thread-System auf programmiersprachlicher Ebene implementieren

- Funktioniert unabhängig vom Betriebssystem
- Sehr leichte Threads (Verwaltungsaufwand: wenige 100 Bytes pro Thread)

Gesucht: Mechanismus, um eine Berechnung nur eine bestimmte Zeitspanne lang laufen zu lassen

Engines

Engines sind ein Modell für die Implementierung von Threads

Eine Engine ist eine Prozedur mit drei Argumenten:

- **ticks** Zahl, die bestimmt wie lange Engine noch laufen darf
- **return** Prozedur, die bestimmt was zu tun ist, wenn die Engine fertig ist bevor die Zeit abgelaufen ist
- **expire** Prozedur, die bestimmt was zu tun ist, wenn die Zeit abgelaufen ist, die Engine aber nicht fertig gerechnet hat

Engines from Continuations Kent Dybvig, Robert Hieb, Journal of Computer Languages 14, 1989

Engines erzeugen

Engines sind durch Funktionen repräsentiert:

```
> (make-engine proc-to-run)
#{Procedure 7819 (unnamed in new-engine)}
```

Rückgabewert ist eine Prozedur, die:

- **ticks**, **return** und **expire** als Argumente nimmt
- wenn aufgerufen, **proc-to-run ticks** lang laufen läßt und entweder **return** oder **expire** aufruft

return und **expire** machen Rückkehr einer Engine zu einem beobachtbarem Ereignis: Verknüpfung mit Scheduler

Engines und Rückkehr

Engines kehren nur durch den Aufruf von `engine-return` zurück

Idee: Gleich die nächste lauffähige Engine starten

Beispiel:

```
(define (make-simple-engine proc)
  (make-engine (lambda ()
                (engine-return (proc))))))
```

```
> (make-simple-engine (lambda () (+ 1 1)))
#{Procedure 7819 (unnamed in new-engine)}
```

Timer (1)

Timer:

- Zeitspanne einstellen
- Bei Ablauf: Aufruf eines `Handlers`
- Zeit wird von Hand weitergezählt: `(decrement-timer)`

```
(define clock 0)
(define handler #f)

(define (start-timer ticks new-handler)
  (set! handler new-handler)
  (set! clock ticks))
```

Timer (2)

```
(define (stop-timer)
  (let ((time-left clock))
    (set! clock 0)
    time-left))

(define (decrement-timer)
  (if (> clock 0)
      (begin
        (set! clock (- clock 1))
        (if (zero? clock)
            (handler))))))
```

Der Scheduler

Der `Scheduler` verwaltet die Threads:

- Queue enthält alle lauffähigen Threads
- Thread aus Queue entnehmen und laufen lassen
- Neue Threads in die Queue stellen

Auswahl des nächsten Threads kann nach unterschiedlichen Kriterien erfolgen (z. B. über Priorität)

Hier: Möglichst einfach; nächster Thread in der Queue; alle Threads rechnen gleich lang

Engines (re-)starten

`ready-queue` enthält Engines

Engine weiterlaufen lassen: Engine neu erzeugen

```
(define ready-queue (make-queue))

(define (start proc)
  (enqueue! ready-queue
            (make-engine
             (lambda ()
               (proc trap)))))

(define (restart k v)
  (enqueue! ready-queue
            (make-engine
             (lambda ()
               (k v)))))
```

Trap (1)

Über `trap` kommunizieren Threads mit dem Scheduler

Der Scheduler bietet drei Dienste an:

- `uninterruptible` Führe eine Prozedur atomar aus
- `start-process` Neuen Thread starten
- `stop-process` Thread beenden

Hinweis: In unserer ersten Implementierung können Engines keine neuen Engines starten

Trap (2)

Implementierung der Dienste:

```
(define (trap message arg)
  (call/cc
   (lambda (k)
     (engine-return
      (lambda ()
        (case message
          ((uninterruptible)
           (restart k (arg)))
          ((start-process)
           (start arg)
           (restart k #f))
          ((stop-process)
           #f))))))))
```

dispatch

`dispatch` läßt eine Engine laufen und ruft wieder `dispatch` auf:

```
(define (dispatch)
  (if (queue-empty? ready-queue)
      'finished
      ((dequeue! ready-queue)
       (time-slice)
       (lambda (trap-handler ticks)
         (trap-handler)
         (dispatch))
       (lambda (engine)
         (enqueue! ready-queue engine)
         (dispatch)))))

(start proc)
(dispatch)
```

Zustand

`active?` läuft momentan eine Engine?

`do-return` und `do-expire` rufen die `return` bzw. `expire` Prozeduren der momentan laufenden Engine auf

```
(define call/cc call-with-current-continuation)
(define active? #f)
(define do-return #f)
(define do-expire #f)
```

Timer starten

`timer-handler` startet den Timer neu

Läuft der Timer ab, wird `do-expire` mit der momentanen Continuation aufgerufen:

```
(define (timer-handler)
  (start-timer (call/cc do-expire)
              timer-handler))
```

`timer-handler` installiert sich selbst wieder als Handler

Die momentane Continuation wird in `do-expire` zur Erzeugung einer neuen Engine verwendet

Konstruktor

`make-engine` ist der Konstruktor für Engines

`proc` ist die Prozedur, die die Engine ausführen soll

```
(define (make-engine proc)
  (new-engine
   (lambda (ticks)
     (start-timer ticks timer-handler)
     (proc)
     (error "engine returned"))))
```

Engines kehren niemals zurück, sondern rufen `engine-return` auf

engine-return

`engine-return` ruft die `return` Funktion der momentan laufenden Engine auf.

```
(define (engine-return value)
  (if active?
      (let ((ticks (stop-timer)))
        (do-return value ticks))
      (error "no engine running")))
```

Engines starten

Um eine Engine zu starten:

- Continuation bei Engine-Start einfangen
- `do-return` auf die `return` Funktion der Engine setzen
- `do-expire` auf die `expire` Funktion der Engine setzen

new-engine

Neue Engine erzeugen, `do-return` und `do-expire` setzen:

```
(define (new-engine resume)
  (lambda (ticks return expire)
    (if active?
      (error "attempt to nest engines")
      (set! active? #t))
    ((call/cc
      (lambda (escape)
        (set! do-return ...)
        (set! do-expire ...)
        (resume ticks))))))
```

- Continuation des Engine-Starts einfangen `escape`
- `((call/cc (lambda (esc) ...)))` ist eine Applikation
- Wird `escape` aufgerufen, muss das Argument ein Thunk sein

do-return setzen

`do-return` setzen:

Continuation für Engine-Start wird ersetzt durch Aufruf von `return`

```
(set! do-return
  (lambda (value ticks)
    (set! active? #f)
    (escape
      (lambda ()
        (return value ticks))))))
```

do-expire setzen

`do-expire` setzen:

- `do-expire` wird vom Handler des Timers aufgerufen
- Neue Engine mit der im `timer-handler` gefangenen Continuation machen

```
(set! do-expire
  (lambda (resume)
    (set! active? #f)
    (escape
      (lambda ()
        (expire (new-engine resume))))))
```

Beispiel (1)

```
(define (time-slice) 2)
(define (print-thread thread-id)
  (lambda (trap)
    (let lp ((n 10))
      (if (zero? n)
          (trap 'stop-process n)
          (begin
             (display (string-append
                       "Thread " (number->string thread-id)))
             (newline)
             (decrement-timer)
             (lp (- n 1))))))))
```

Beispiel (2)

```
(define (two-threads)
  (round-robin-scheduler
   (lambda (trap)
     (trap 'start-process (print-thread 1))
     (trap 'start-process (print-thread 2))
     (trap 'stop-process #f))))
```