

# Linguagem de Programação



**LabLua – PUC-Rio**

[www.lua.inf.puc-rio.br](http://www.lua.inf.puc-rio.br)

*Francisco Sant'Anna  
Noemi Rodrigues  
Roberto Ierusalimschy*

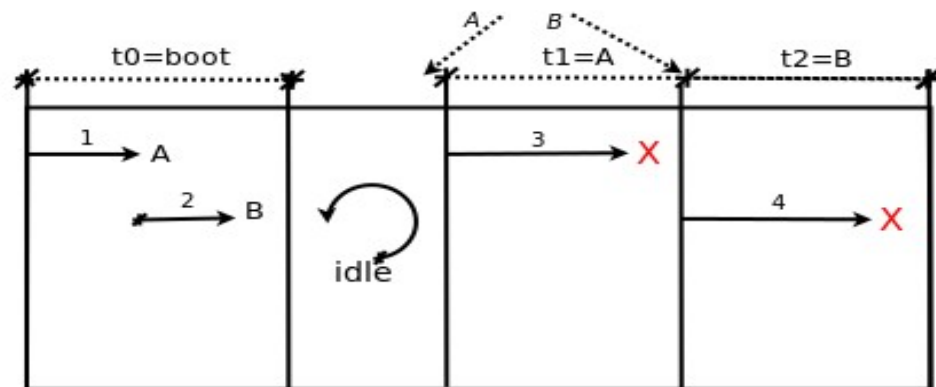
# Características

1. Execução síncrona-reativa
2. Composições
3. Compartilhamento de memória
4. Eventos internos
5. Integração com C
6. Escopos e Finalização
7. Temporizadores
8. Execução Assíncrona
9. Abstrações

# 1. Modelo de Execução

1. O programa inicia na “reação de *boot*” em apenas uma trilha.
2. Trilhas ativas executam até esperarem ou terminarem. Esse passo é conhecido como “reação” e sempre executa em tempo *bounded*.
3. A ocorrência de um evento de entrada acorda **todas** as trilhas esperando aquele evento. Repete o “passo 2”.

```
par/and do
  <...>      // 1
  await A;
  <...>      // 3
with
  <...>      // 2
  await B;
  <...>      // 4
end
```



<...> são segmentos de trilha que não esperam  
(e.g. atribuições, chamadas de função)

# 1. Modelo de Execução

- Céu assegura execução “bounded”
  - Todos os loops devem ter `await` ou terminar

```
loop do
  if <cond> then
    break;
  end
end
```

```
loop do
  if <cond> then
    break;
  else
    await A;
  end
end
```

- *Limitação: operações longas*

## 2. Composições

- Programação reativa estruturada
  - *sequência, repetição e paralelismo*
- Abortamento ortogonal
  - par/or
  - Possibilidade de terminar uma linha de execução “por fora”

## 2. Composições

- Piscando LEDs

- 1. *on  $\leftrightarrow$  off a cada 500ms*

- 2. *Parar após 5s*

- 3. *Repetir após 2s*

- Composições

- par, seq, loop
  - ~~variáveis de estado~~
  - inferência

```
loop do
  par/or do
    loop do
      await 500ms;
      _leds_toggle();
    end
  with
    await 5s;
  end
  await 2s;
end
```

# par/and, par/or

```
loop do
  par/and do
    ...
  with
    await 1s;
  end
end
// PERIODIC ("at least")
```

```
loop do
  par/or do
    ...
  with
    await 1s;
  end
end
// TIMEOUT ("at most")
```

# 3. Compartilhamento de Memória

```
var int x=1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
End
_printf("%d\n", x);
```

```
var int x=1;
par/and do
    await A;
    x = x + 1;
with
    await A;
    x = x * 2;
End
_printf("%d\n", x);
```

## ■ Análise estática

- `ceu --safety 0` # permite ambos
- `ceu --safety 1` # permite ESQ, recusa DIR
- `ceu --safety 2` # recusa ambos



# Exercício 1 (revisão)

```
int state = 1;
unsigned long old;

void setup () {
    old = millis();
    digitalWrite(LED_PIN, state);
}

void loop () {
    unsigned long now = millis();
    if (now >= old+1000) {
        <X = X + 1>
        digitalWrite(LED_PIN, state);
    }
    ;
    int but = digitalRead(BUT_PIN);
    if (but == HIGH) {
        <X = X * 2>
        while(1);
    }
}
```

```
Thread* t1;
void Thread1 (void) {
    while (TRUE) {
        digitalWrite(LED_PIN, HIGH);
        chThdSleepMilliseconds(1000);
        if (chThdShouldTerminate())
            break;
        <X = X + 1>
        digitalWrite(LED_PIN, LOW);
        chThdSleepMilliseconds(1000);
        if (chThdShouldTerminate())
            break;
    }
}

void Thread2 (void) {
    while (TRUE) {
        int but = digitalRead(BUT_PIN);
        if (but == HIGH) {
            <X = X * 2>
            chThdSleepMilliseconds(1000);
            break;
        }
    }
}

void main () {
    t1 = chThdCreateStatic(..., Thread1);
    chThdSleepMilliseconds(10000);
    chThdTerminate(t1);
    chThdCreateStatic(..., Thread2);
}
```

# 4. Eventos internos

- Mecanismo interno de sinalização
- Emissão do próprio programa
  - (*vs ambiente*)
- Múltiplos numa mesma reação
  - (*vs único*)
- Execução por pilha
  - (*vs fila*)

# 4. Eventos internos

- Permite a criação de novos mecanismos de controle (e.g. subrotinas, exceções)

```
event int* inc;  
par do  
    // define subrotina  
    loop do  
        var int* p = await inc;  
        *p = *p + 1;  
    end  
with  
    // use subrotina  
    <...>  
    var int v = 1;  
    emit inc => &v;    // stack this continuation  
    _assert(v == 2);  
end
```

# 5. Integração com C

- Sintaxe diferenciada (“\_”)

```
native do  
  #include <assert.h>  
  int I = 0;  
  int inc (int i) {  
    return I+i;  
  }  
end  
_assert(_inc(_I));
```

- “C hat” (execução não segura)
- Sem análise de execução “bounded”
- E em relação a efeitos colaterais?

# 5. Integração com C

- Anotações **pure** and **safe**

```
pure _inc();  
safe _f() with _g();  
  
par do  
  _f(_inc(10));  
with  
  _g();  
end
```

# 6. Escopos locais & Finalização

```
par/or do
  var _message_t msg;
  <...> // prepare msg
  _send_request(&msg);
  await SEND_ACK;
with
  <...>
end
```

local memory  
to  
external pointer

May terminate

line 5 : call to "`_send_request`"  
requires ``finalize``

```
par/or do
  var _FILE* f = _fopen(...);
  _fwrite(..., f);
  await 1s;
  _fwrite(..., f);
  await 1s;
  _fclose(f);
with
  <...>
end
```

external memory  
to  
local pointer

May terminate

line 2 : attribution requires  
``finalize``

# 6. Escopos locais & Finalização

```
par/or do
  var _message_t msg;
  <...> // prepare msg
  finalize
    _send_request(&msg);
  with
    _send_cancel(&msg);
  end
  await SEND_ACK;
with
  <...>
end
```

```
par/or do
  var _FILE* f;
  finalize
    f = _fopen(...);
  with
    _fclose(f);
  end
  _fwrite(..., f);
  await 1s;
  _fwrite(..., f);
  await 1s;
  _fclose(f);
with
  <...>
end
```

# 7. Temporizadores

- Muito comuns em aplicações reativas
  - *sampling, timeouts*
- **await** suporta tempo (i.e., *ms, min*)
  - ... e compensa atrasos do sistema

```
await 2ms;  
v = 1;  
await 1ms;  
v = 2;
```

- 5ms elapse
- late = 3ms
- late = 2ms

```
par/or do  
    await 10ms;  
    <...>    // no awaits  
    await 1ms;  
    v = 1;  
with  
    await 12ms;  
    v = 2;  
end
```



# 8. Execução Assíncrona

- Operações longas
- Paralelismo real
- Garantias de “*hard real-time*”
- Geração de *input*

# 9. Abstrações / Organismos

```
var SDL_Rect r1;
```

```
var SDL_Rect r2;
```

```
par do
```

```
  loop do
```

```
    loop i in 100 do
```

```
      await 10ms;
```

```
      r1.x = r1.x + 1;
```

```
    end
```

```
    loop i in 100 do
```

```
      await 10ms;
```

```
      r1.y = r1.y + 1;
```

```
    end
```

```
    loop i in 100 do
```

```
      await 10ms;
```

```
      r1.x = r1.x - 1;
```

```
    end
```

```
    loop i in 100 do
```

```
      await 10ms;
```

```
      r1.y = r1.y - 1;
```

```
    end
```

```
  end
```

```
with
```

```
  loop do
```

```
    loop i in 100 do
```

```
      await 15ms;
```

```
      r2.x = r2.x + 1;
```

```
    end
```

```
    loop i in 100 do
```

```
      await 15ms;
```

```
      r2.y = r2.y + 1;
```

```
    end
```

```
    loop i in 100 do
```

```
      await 15ms;
```

```
      r2.x = r2.x - 1;
```

```
    end
```

```
    loop i in 100 do
```

```
      await 15ms;
```

```
      r2.y = r2.y - 1;
```

```
    end
```

```
  end
```

```
with
```

```
  every SDL_REDRAW do
```

```
    SDL_RenderFillRect(ren, &r1);
```

```
    SDL_RenderFillRect(ren, &r2);
```

```
  end
```

```
end
```

r1

r2

# Organismos

- *Organismo ~ (Objeto + Thread)*

```
class <ID> with  
  <interface>  
    // properties  
    // methods  
do  
  <body>  
    // Céu code  
end
```

```
class Rect with  
  var _SDL_Rect r;  
  var int dt;  
do  
  par do  
    loop do  
      loop i in 100 do  
        await (this.dt)ms;  
        this.r.x = this.r.x + 1;  
      end  
      loop i in 100 do  
        await (this.dt)ms;  
        this.r.y = this.r.y + 1;  
      end  
      loop i in 100 do  
        await (this.dt)ms;  
        this.r.x = this.r.x - 1;  
      end  
      loop i in 100 do  
        await (this.dt)ms;  
        this.r.y = this.r.y - 1;  
      end  
    end  
  end  
  with  
    every SDL_REDRAW do  
      _SDL_RenderFillRect(ren, &this.r);  
    end  
  end  
end
```

```
class Rect with  
    <...>  
do  
    <...>  
end
```

```
var Rect r1 with  
    this.r.* = *;  
    this.dt  = 10;  
end;
```

```
var Rect r2 with  
    this.r.* = *;  
    this.dt  = 15;  
end;
```

```
await FOREVER;
```

# Demo...

- Vetores
- Colisões
- Escopo léxico
  - Dados e corpo são abortados
  - Abortamento ortogonal / Finalização
- Instâncias dinâmicas
  - Pools e limites
  - Término do corpo
  - Escopo léxico