Modelos de Concorrência

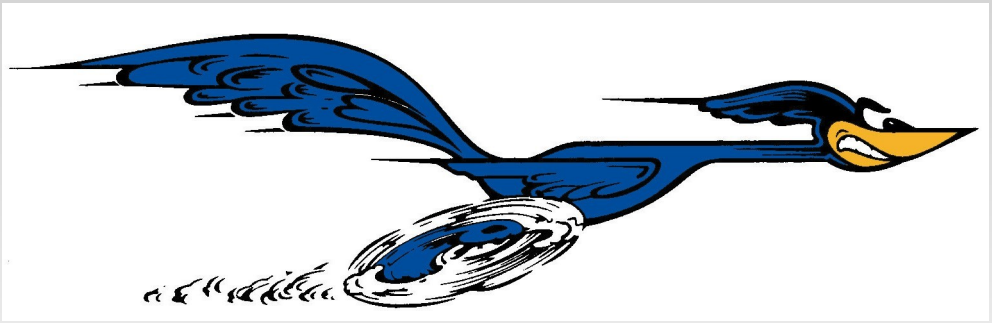
LabLua – PUC-Rio
www.lua.inf.puc-rio.br

Exercício 1 (revisão)

- Piscar o LED a cada 1 segundo
- Parar ao pressionar o botão, mantendo o LED aceso para sempre

```
void loop () {
    digitalWrite(LED_PIN, HIGH);
    delay(1000);
    digitalWrite(LED_PIN, LOW);
    delay(1000);

    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        while(1);
    }
}
```

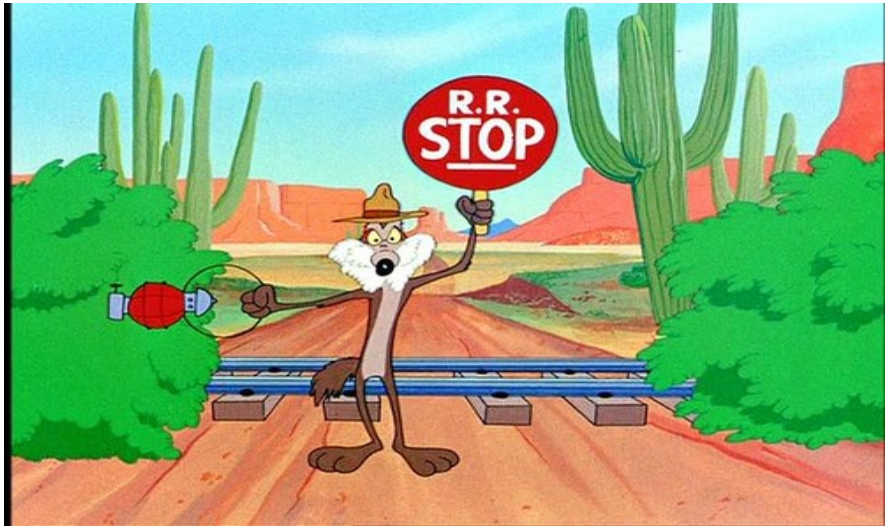
- Programa interativo!



Programa Reativo

Seu pior inimigo?

Chamadas Bloqueantes



Exercício 1 (revisão)

- Guardar *timestamp* da última mudança
- Guardar estado atual do LED

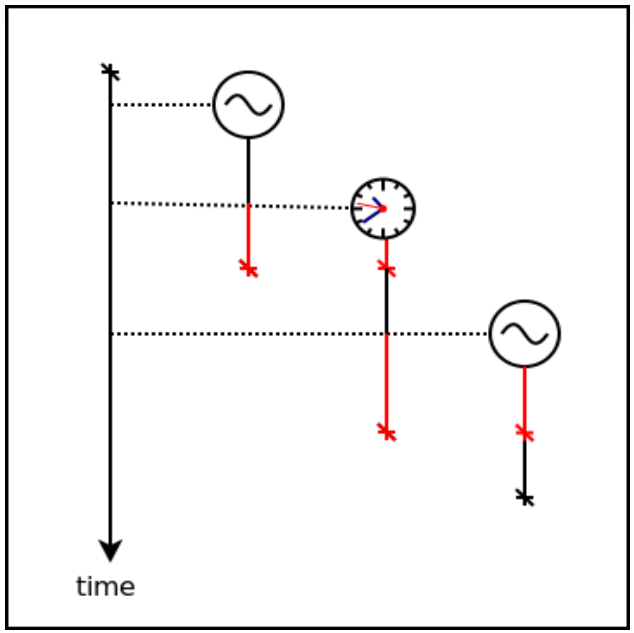
```
int state = 1;
unsigned long old;
void setup () {
    old = millis();
    digitalWrite(LED_PIN, state);
}
void loop () {
    unsigned long now = millis();
    if (now >= old+1000) {
        old = now;
        state = !state;
        digitalWrite(LED_PIN, state);
    }

    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        while(1);
    }
}
```

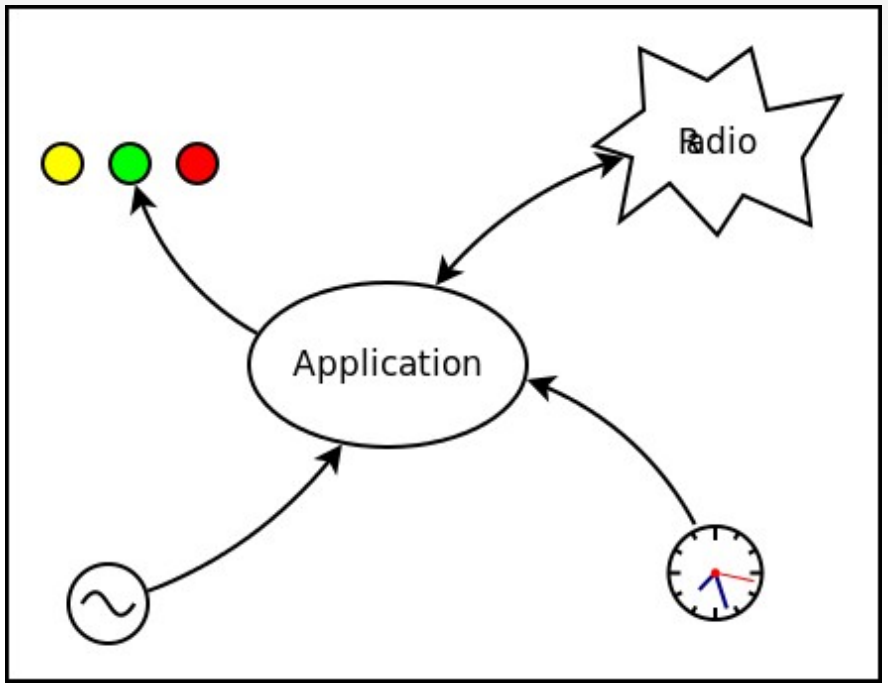
```
void loop () {
   unsigned long now = millis();
   if (now >= old+1000) {
      old = now;
      state = !state;
      digitalWrite(LED_PIN, state);
   }
}
```

```
void loop () {
   int but = digitalRead(BUT_PIN);
   if (but) {
      digitalWrite(LED_PIN, HIGH);
      exit();
   }
}
```

Concorrência

- O que é concorrente com o que?
- Cada estímulo gera uma reação
  - Estímulo é infinitesimal
  - Reação tem duração





- *Reação "A" é concorrente com reação "B"*

E daí?

- Problemas quando reações concorrentes acessam o mesmo recurso.

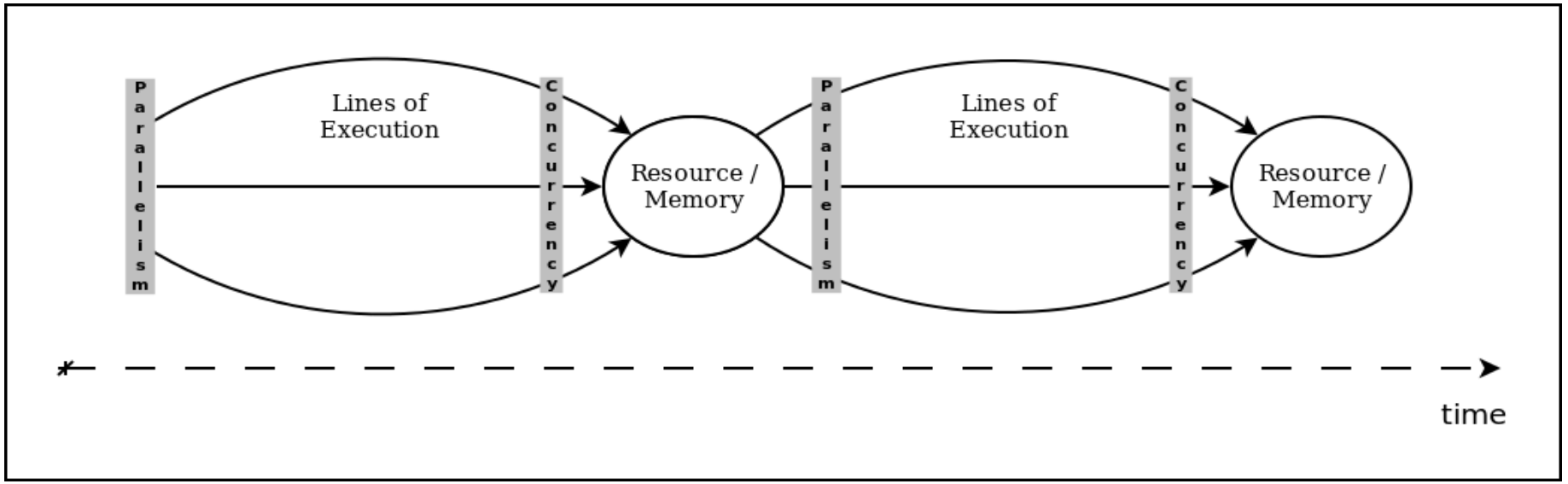
```
int state = 1;
unsigned long old;
void setup () {
    old = millis();
    digitalWrite(LED_PIN, state);
}
void loop () {
    unsigned long now = millis();
    if (now >= old+1000) {
        old = now;
        state = !state;
        digitalWrite(LED_PIN, state);
    }

    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        while(1);
    }
}
```

```
void loop () {
    unsigned long now = millis();
    if (now >= old+1000) {
        old = now;
        state = !state;
        digitalWrite(LED_PIN, state);
    }
}
```

```
void loop () {
    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        exit();
    }
}
```

Concorrência vs Paralelismo

- Concorrência
  - Acessos simultâneos (ao **mesmo** recurso)
- Paralelismo
  - Execuções simultâneas (em **múltiplas** linhas)



Artigos & Videos - 02

- Concorrência e Paralelismo
  - Rob Pike - *"Concurrency Is Not Parallelism"*
  - Wikipedia - *"Embarrassingly Parallel"*

)

Modelos de Execução Concorrente

- Por quê?
  - Como descrever e entender as partes de um sistema concorrente (e.g., atividades, processos, atores, etc.).
  - Vocabulário e semântica
    - execução (escalonamento)
    - composição
    - compartilhamento
    - comunicação
    - sincronização

Modelos de Execução Concorrente

- Modelo Assíncrono
  - Execução independente / Sincronização explícita
    - *Threads* + *locks/mutexes* (*p-threads*, *Java-Threads*)
    - Atores + troca de mensagens (*Erlang*, *Go*)
- Modelo Síncrono
  - Execução dependente / Sincronização implícita
    - *Arduino-Loop*, *Game-Loop*, Padrão *Observer*, Circuitos

Modelo Assíncrono

- Execução independente
  - Arduino: *ChibiOS*

```
void Thread1 (void) {
  <...>
}

void Thread2 (void) {
  <...>
}

void setup() {
  chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

Exercício 1 (async)

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

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

void loop () {
    unsigned long now = millis();
    if (now >= old+1000) {
        old = now;
        state = !state;
        digitalWrite(LED_PIN, state);
    }

    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        while(1);
    }
}
```

```
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
  }
}

void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      break;
    }
  }
}

void setup () {
  chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

Exercício 1 (async)

```
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
  }
}

void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      break;
    }
  }
}

void setup () {
  chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

```
Thread* t1;

void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
  }
}

void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      break;
    }
  }
}

void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

Artigos & Videos - 03

- Terminação de Threads
  - Java - "*Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?*"
  - pthreads – `man pthreads_cancel`
  - ChibiOS - *"How to cleanly stop the OS"*

Exercício 1 (async)

```
Thread* t1;

void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
  }
}

void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      break;
    }
  }
}

void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

```
Thread* t1;

void Thread1 (void) {
  while (!chThdShouldTerminate()) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
  }
}

void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      break;
    }
  }
}

void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

Exercício 1 (async)

```
Thread* t1;

void Thread1 (void) {
  while (!chThdShouldTerminate()) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
  }
}

void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      break;
    }
  }
}

void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

```
Thread* t1;
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
  }
}
void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      break;
    }
  }
}
void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

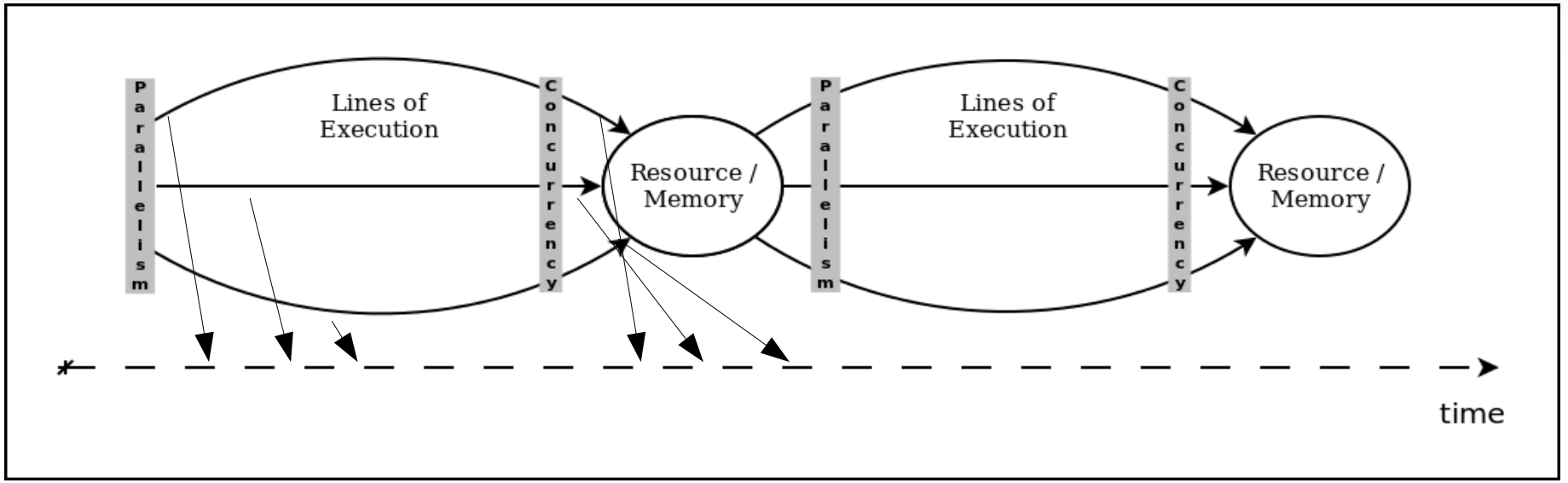
Exercício 1 (async)

```
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
  }
}
void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      break;
    }
  }
}
void setup () {
  chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

```
Thread* t1;
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
  }
}
void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      break;
    }
  }
}
void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

Escalonamento

- Como as atividades dividem o tempo de CPU
  - ex., Preempção, Cooperação, Prioridade, “Batch”

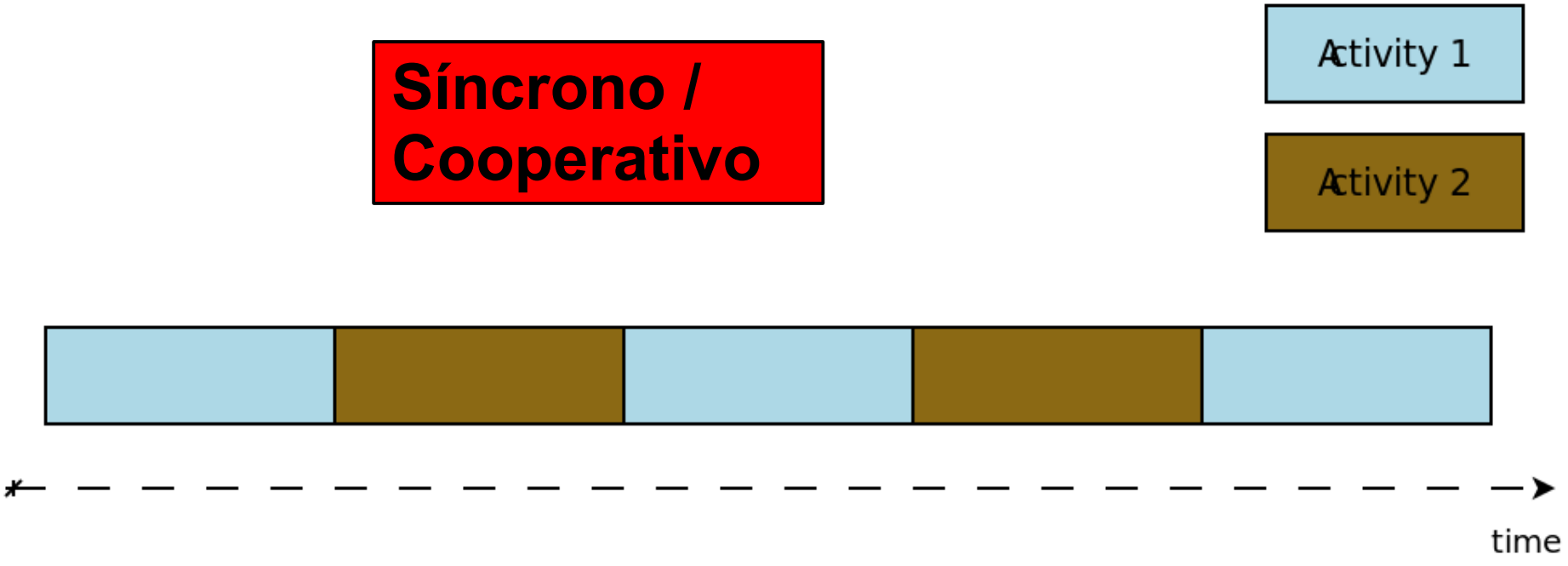


Exercício 1 (async)

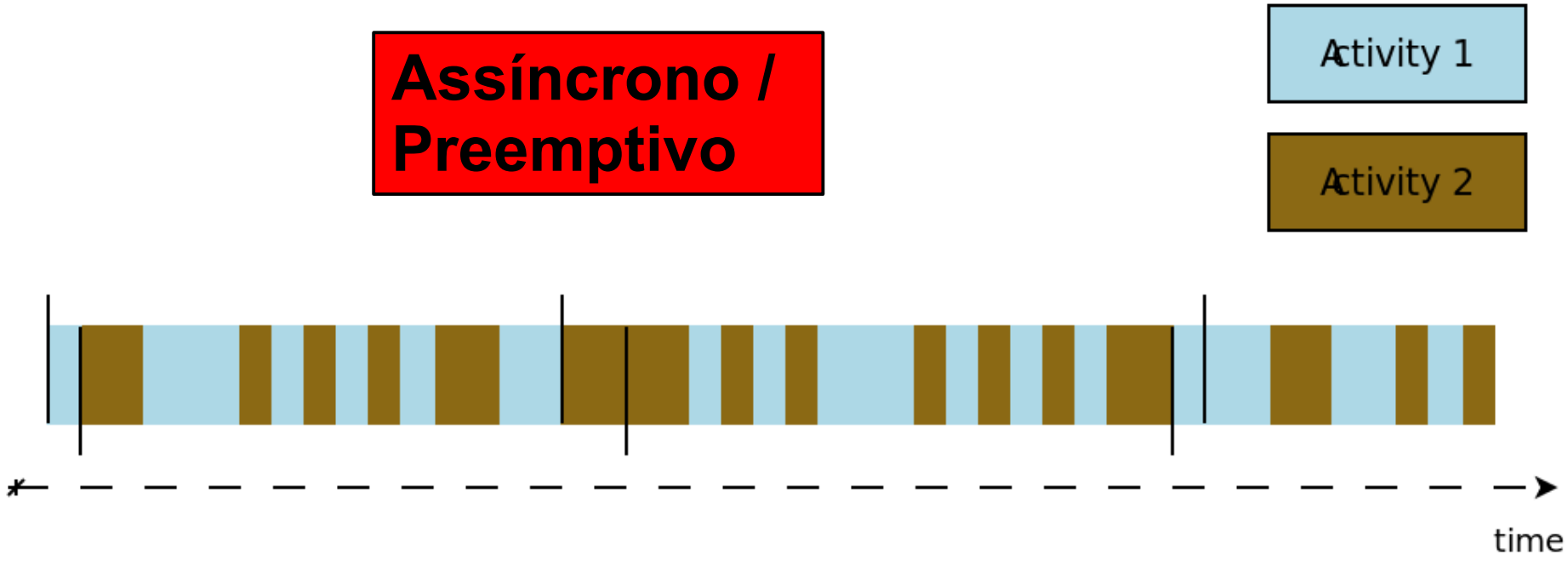
```
int state = 1;
unsigned long old;

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

void loop () {
    unsigned long now = millis();
    if (now >= old+1000) {
        old = now;
        state = !state;
        digitalWrite(LED_PIN, state);
    }

    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        while(1);
    }
}
```

```
Thread* t1;
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
  }
}
void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      break;
    }
  }
}
void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

Síncrono / Cooperativo



Assíncrono / Preemptivo



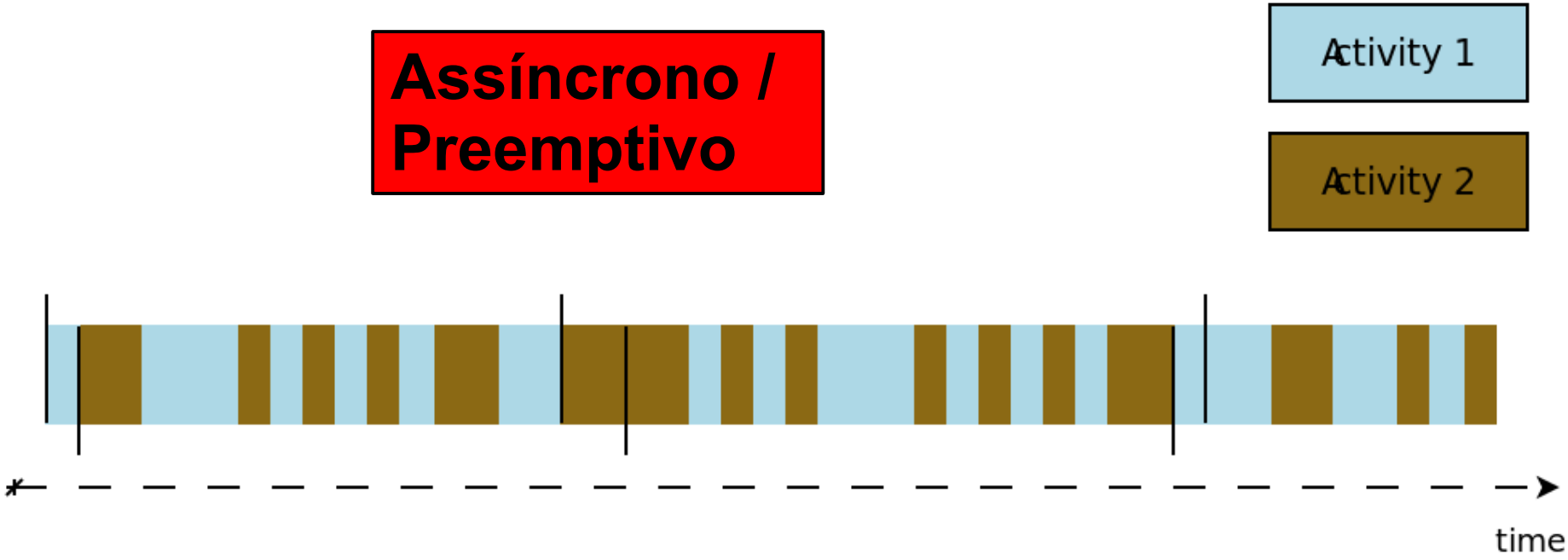
Exercício 1 (async)

```
Thread* t1;
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
  }
}
void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      break;
    }
  }
}
void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

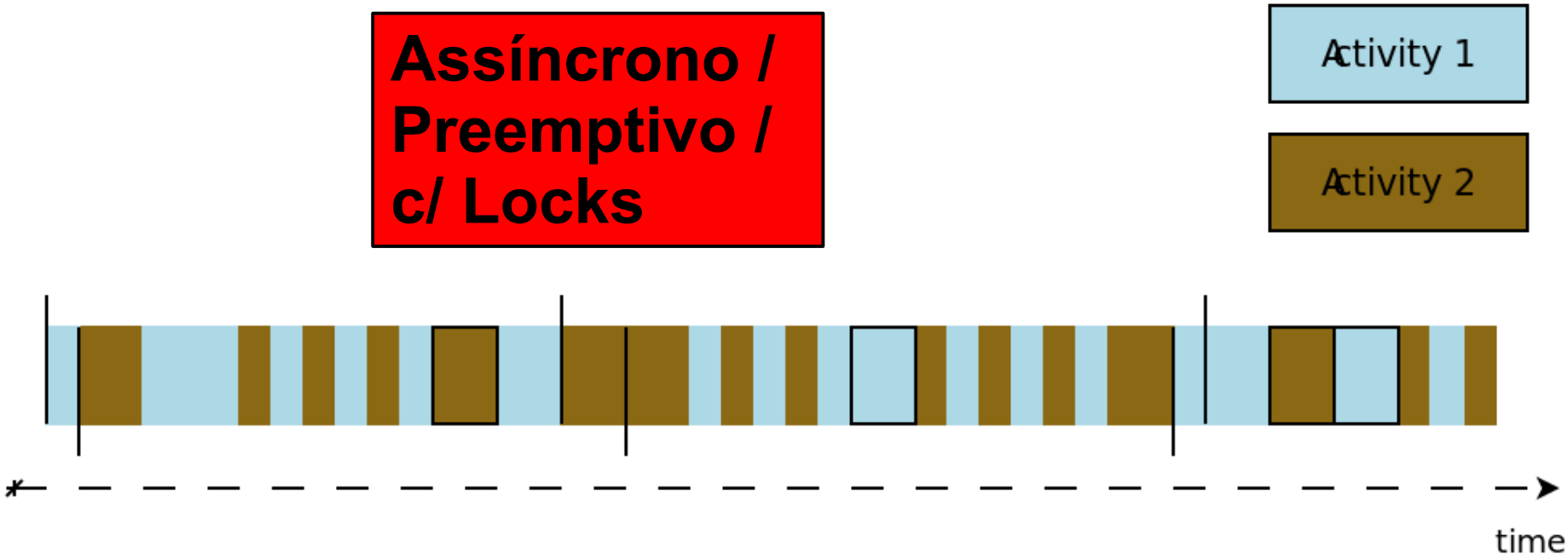
Condição de Corrida

```
MUTEX_DECL(mut);
Thread* t1;
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    chMtxLock(&mut);
    if (chThdShouldTerminate())
        break;
    digitalWrite(LED_PIN, LOW);
    chMtxUnlock(&mut);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
  }
}
void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      chMtxLock(&mut);
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      chMtxUnlock(&mut);
      break;
    }
  }
}
void setup () {
```

Assíncrono / Preemptivo



Assíncrono / Preemptivo / c/ Locks



Exercício 1 (async)

```
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    digitalWrite(LED_PIN, LOW);
    chThdSleepMilliseconds(1000);
  }
}
void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      break;
    }
  }
}
void setup () {
  chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

12 das 32 linhas para sincronização (37%)

```
MUTEX_DECL(mut);
Thread* t1;
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    chThdSleepMilliseconds(1000);
    chMtxLock(&mut);
    if (chThdShouldTerminate())
        break;
    digitalWrite(LED_PIN, LOW);
    chMtxUnlock(&mut);
    chThdSleepMilliseconds(1000);
    if (chThdShouldTerminate())
        break;
  }
}
void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      chMtxLock(&mut);
      digitalWrite(LED_PIN, HIGH);
      chThdTerminate(t1);
      chMtxUnlock(&mut);
      break;
    }
  }
}
void setup () {
  t1 = chThdCreateStatic(..., Thread1);
  chThdCreateStatic(..., Thread2);
}
```

Exemplo – "Cálculo Pesado"

- *ordenação, criptografia, compressão, codificação/conversão*
- Piscar o LED a cada ~~1 segundo~~ término de uma operação longa
- Parar ao pressionar o botão, mantendo o LED aceso para sempre

```
void loop () {
    digitalWrite(LED_PIN, HIGH);
    f();
    digitalWrite(LED_PIN, LOW);
    f();

    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        while(1);
    }
}
```

```
void Thread1 (void) {
  while (TRUE) {
    digitalWrite(LED_PIN, HIGH);
    f();  // operação longa
    digitalWrite(LED_PIN, LOW);
    f();  // operação longa
  }
}

void Thread2 (void) {
  while (TRUE) {
    int but = digitalRead(BUT_PIN);
    if (but) {
      digitalWrite(LED_PIN, HIGH);
      stop();
    }
  }
}
```

Exercício – “Cálculo Pesado”

- Criptografia XTEA
  - https://en.wikipedia.org/wiki/XTEA
- Criptografar / Descriptografar uma string
- Calcular o tempo de execução das operações:
  - Medir o tempo de resposta para 10 execuções
- Variar o tamanho da string e avaliar a “responsividade”

```
void loop () {
    digitalWrite(LED_PIN, HIGH);
    encipher(...);
    digitalWrite(LED_PIN, LOW);
    decipher(...);

    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        while(1);
    }
}
```

```
// XTEA

void encipher (...) {
    <...>
}

void decipher (...) {
    <...>
}
```

Exercício – "Cálculo Pesado"

- Usar Serial para debug/output
  - Remover output na hora de medir!
- Exemplo: *03_calc.ino*

```
void setup () {
    Serial.begin(9600);
}
void loop () {
    unsigned long t1 = millis();
    Serial.print(<v>);    // antes
    encipher(<v>);
    Serial.print(<v>);    // durante
    decipher(<v>);
    Serial.print(<v>);    // depois
    unsigned long t2 = millis();
    Serial.print(t2-t1);  // total
}
```

```
#include "xtea.c"

#define LED 13

void setup () {
    pinMode(LED, OUTPUT);
    Serial.begin(9600);
}

int led = 0;

uint32_t key[] = { 1, 2, 3, 4 };
uint32_t v[]   = { 10, 20 };

void loop () {
    led = !led;
    digitalWrite(LED, led);

    unsigned long t1 = millis();

    Serial.print("antes: ");
        Serial.print(v[0]);
        Serial.print(" ");
        Serial.println(v[1]);

    encipher(32, v, key);

    Serial.print("durante: ");
        Serial.print(v[0]);
        Serial.print(" ");
        Serial.println(v[1]);

    decipher(32, v, key);

    Serial.print("depois: ");
        Serial.print(v[0]);
        Serial.print(" ");
        Serial.println(v[1]);

    unsigned long t2 = millis();
    Serial.println(t2-t1);
}
```

Exercício – “Cálculo Pesado”

- O que fazer se a execução demora demais?
  - sistema não mais reativo
- Inversão de controle
  - re-implementar o algoritmo!
- Usar *threads*
  - Praticamente não há concorrência

Analogia com Video Games

- *colisão, path finding, animações, renderização, etc.*
- FPS: frames por segundo

```
void loop () {
    INPUT();
    ia();
    colisao();
    renderizacao();
    OUTPUT();
}
// FPS: quantas vezes o “loop”
//      executa a cada segundo
```

Modelo Síncrono

- Execução dependente
  - Loop do Arduino

Iteração do Loop:
- instante
- tick
- tempo lógico

```
int state = 1;
unsigned long old;
void setup () {
    old = millis();
    digitalWrite(LED_PIN, state);
}
void loop () {
    unsigned long now = millis();
    if (now >= old+1000) {
        old = now;
        state = !state;
        digitalWrite(LED_PIN, state);
    }

    int but = digitalRead(BUT_PIN);
    if (but) {
        digitalWrite(LED_PIN, HIGH);
        while(1);
    }
}
```

```
void loop () {
   unsigned long now = millis();
   if (now >= old+1000) {
      old = now;
      state = !state;
      digitalWrite(LED_PIN, state);
   }
}
```

```
void loop () {
   int but = digitalRead(BUT_PIN);
   if (but) {
      digitalWrite(LED_PIN, HIGH);
      exit();
   }
}
```

Modelo Síncrono

- Durante uma unidade de tempo lógico, o ambiente está invariante e não interrompe o programa
- Implementação:
  - *Sampling*: Arduino Loop*
  - *Event-driven*: Padrão *Observer*

```
foreach TICK do
   read_inputs();
   react();
end
```

```
wait ANY_EVENT_CHANGE do
   react();
end
```

- **Hipótese de sincronismo:** "*Reações executam infinitamente mais rápido do que a taxa de eventos.*"

Padrão Observer

- “Hollywood principle: don't call us, we'll call you.”
- Ocorrência de um evento executa uma “callback” no código
  - Botão => button_changed()
  - Timer => timer_expired()
  - Rede => packet_received()

Hello World: input

- Fazer o LED acompanhar o estado do botão

```
#define LED_PIN 13
#define BUT_PIN  2

void setup () {
    pinMode(LED_PIN, OUTPUT);
    pinMode(BUT_PIN, INPUT);
}

void loop () {
    int but = digitalRead(BUT_PIN);
    digitalWrite(LED_PIN, but);
}
```

```
#include "event_driven.c"

#define LED_PIN 13
#define BUT_PIN  2

void button_changed (int pin, int v) {
    digitalWrite(LED_PIN, v);
}

void init () {
    button_listen(BUT_PIN);
}
```

Tarefa 3

(a conferir no início da próxima aula)

- Implementar “event_driven.c”
  - Tratador para botões
  - 2 timers

- Reimplementar os exemplos com orientação a eventos:
  - Hello World: Input
  - Tarefa 2

Tarefa 3 - API

```
/* Funções de registro: */

void button_listen (int pin) {
   <...>                    // “pin” passado deve gerar notificações
}

void timer_set (int ms) {
   <...>                    // timer deve expirar após “ms” milisegundos
}

/* Callbacks */

void button_changed (int pin, int v);  // notifica que “pin” mudou para “v”
void timer_expired (void);             // notifica que o timer expirou

/* Programa principal: */

void setup () {
    <...>                   // inicialização da API
    init();                 // inicialização do usuário
}

void loop () {
    <...>                   // detecta novos eventos
    button_changed(...);    // notifica o usuário
    <...>                   // detecta novos eventos
    timer_expired(...);     // notifica o usuário
}
```

Modelos de Execução Concorrente

- Modelo Assíncrono
  - Execução independente
  - Sincronização explícita
  - Concorrência *fine-grained*
  - Chamadas bloqueantes / demoradas

- Modelo Síncrono
  - Execução dependente
  - Sincronização implícita
  - Concorrência *coarse-grained*
  - Hipótese de sincronismo