

Sistemas embarcados

Projeto de software de
sistemas embarcados
com
protothreads

Threads x eventos

```
graph TD; A([camada de software do sistema]) --> B[Threads]; A --> C[Eventos];
```

camada de software do sistema

Threads

Eventos

Execução de uma aplicação através da **divisão** da mesma em **duas ou mais tarefas** executadas **concorrentemente** (*multithreading*).

O gerenciamento e a **execução de threads** é uma função básica de um **sistema operacional** (SO)

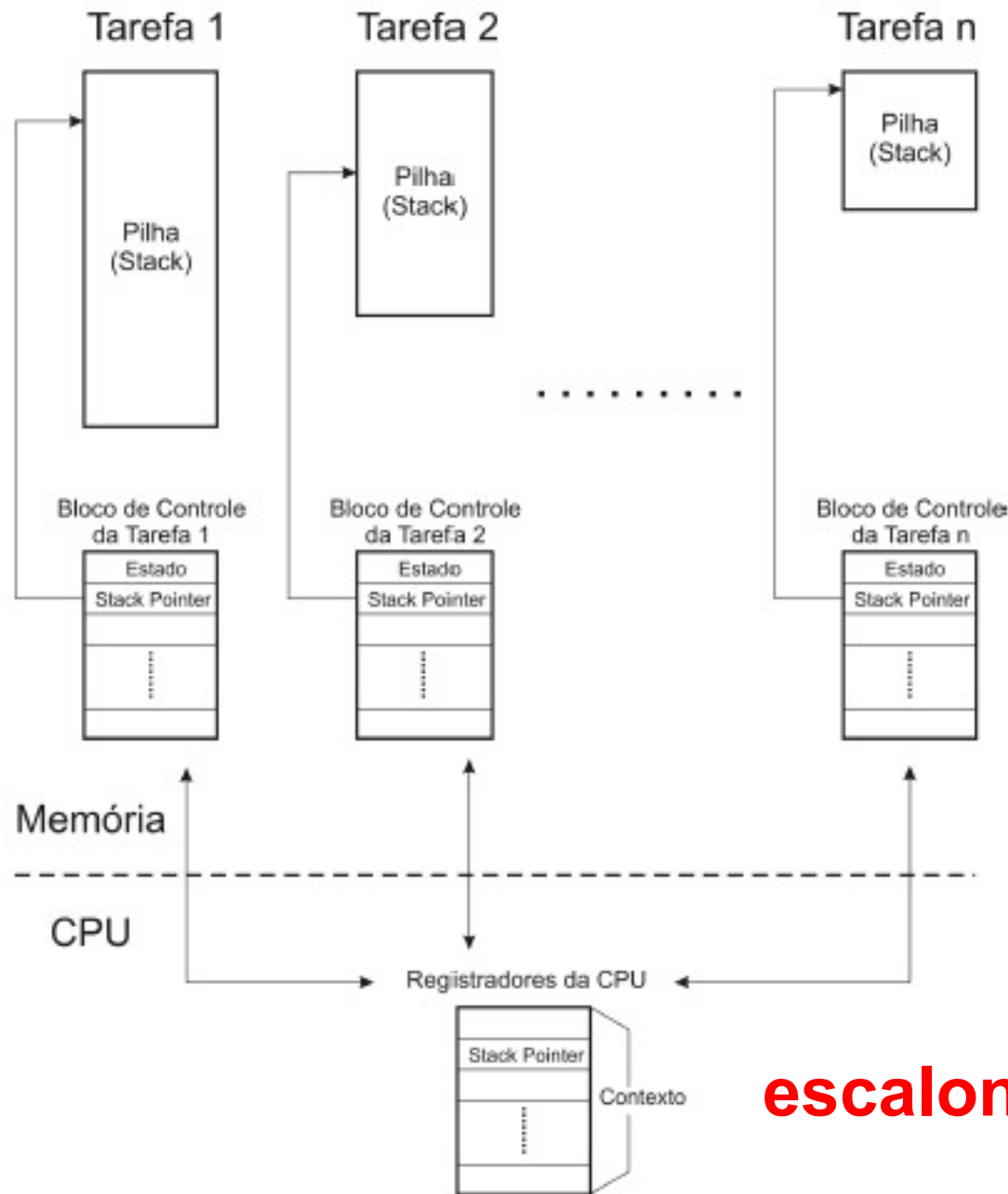
Em um **sistema guiado por eventos** as tarefas são compostas por **estados** do sistema, onde a **transição entre estados** é disparada por eventos.

Sistemas **guiados por eventos** são modelados como **máquinas de estados finitos** (FSM)

Threads x eventos

	Eventos/FSM	Threads
Lógica, controle de fluxo	Precisa ser organizado como máquina de estados. Pode ser mais difícil de entender a lógica.	Usa programação estruturada , cada thread é um programa independente* dos demais.
Manutenção	Mais difícil de manter conforme mais estados são acrescentados	Manutenção mais fácil (mas depende das interdependências).
Memória	Usa um única pilha, tende a usar menos memória RAM .	Cada thread tem uma pilha exclusiva. Usa mais RAM .
Depuração	Mais difícil . Necessário acompanhar transições de estados.	Mais fácil . Retém na pilha a sequência de chamadas de funções.
Testes	Pode se garantir o funcionamento correto , testando todas as sequências de eventos/estados.	Interdependências entre threads podem causar “bugs” de difícil detecção e correção .
Preempção e contexto.	Não possui . Contexto não é preservado.	Sim . Pode realizar preempção e manter o contexto.
Tempo real	Difícil de projetar.	Mais fácil de projetar. Técnicas de escalonamento bem conhecidas.

Projeto com threads



**pilhas
exclusivas !**

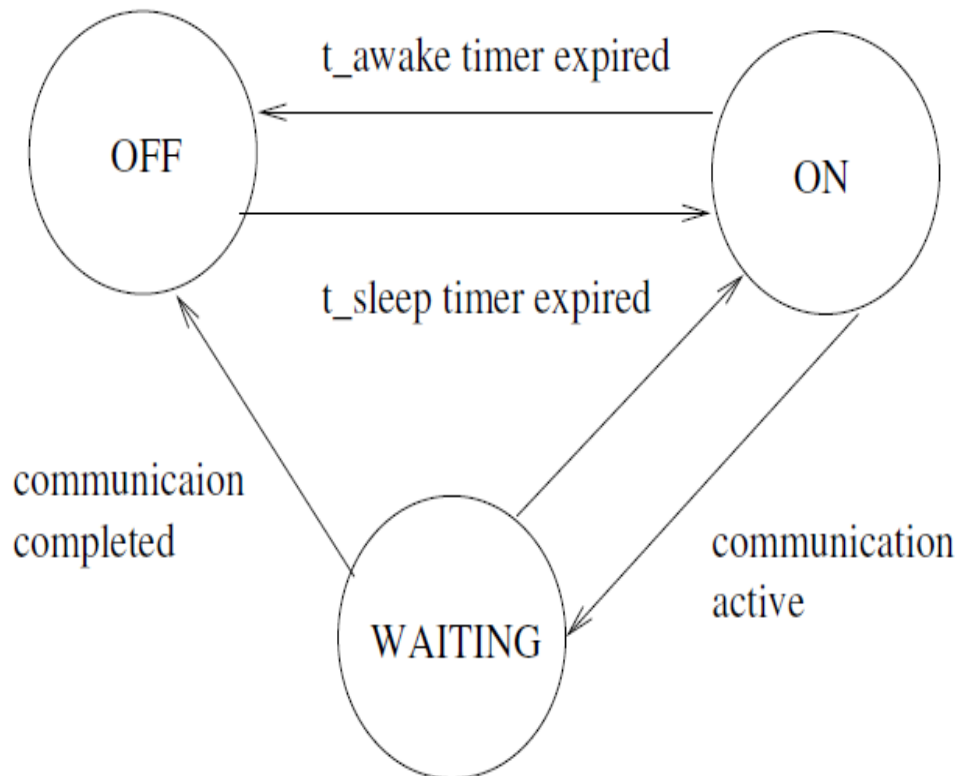
↙
Mais RAM!

blocos de controle

escalonamento

Projeto com eventos/FSM

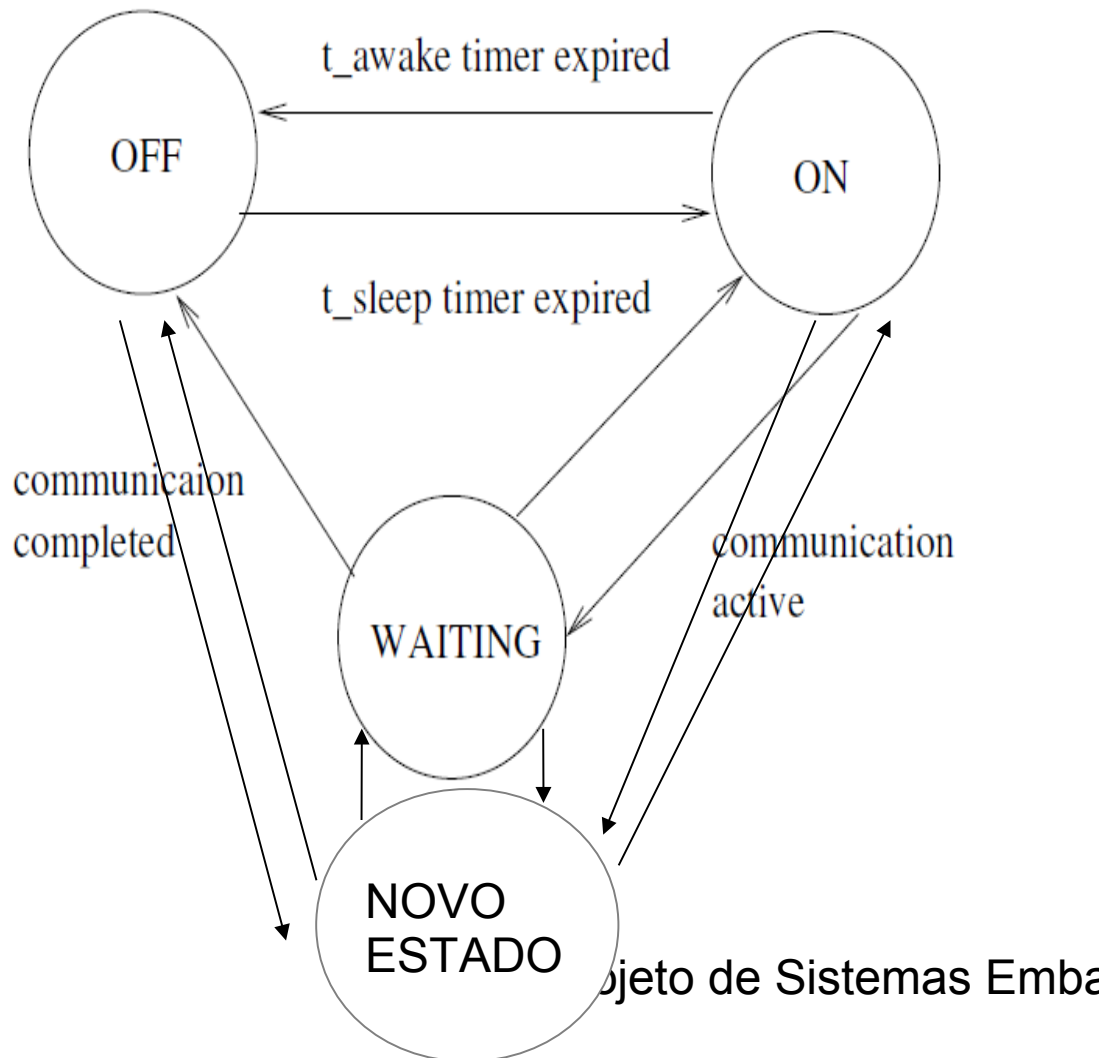
**Complexidade aumenta
muito a cada novo estado!**



```
enum {  
    ON,  
    WAITING,  
    OFF  
} state;  
  
void radio_wake_eventhandler() {  
    switch(state) {  
  
        case OFF:  
            if(timer_expired(&timer)) {  
                radio_on();  
                state = ON;  
                timer_set(&timer, T_AWAKE);  
            }  
            break;  
  
        case ON:  
            if(timer_expired(&timer)) {  
                timer_set(&timer, T_SLEEP);  
                if(!communication_complete()) {  
                    state = WAITING;  
                } else {  
                    radio_off();  
                    state = OFF;  
                }  
            }  
            break;  
  
        case WAITING:  
            if(communication_complete()  
                || timer_expired(&timer)) {  
                state = ON;  
                timer_set(&timer, T_AWAKE);  
            } else {  
                radio_off();  
                state = OFF;  
            }  
            break;  
    }  
}
```

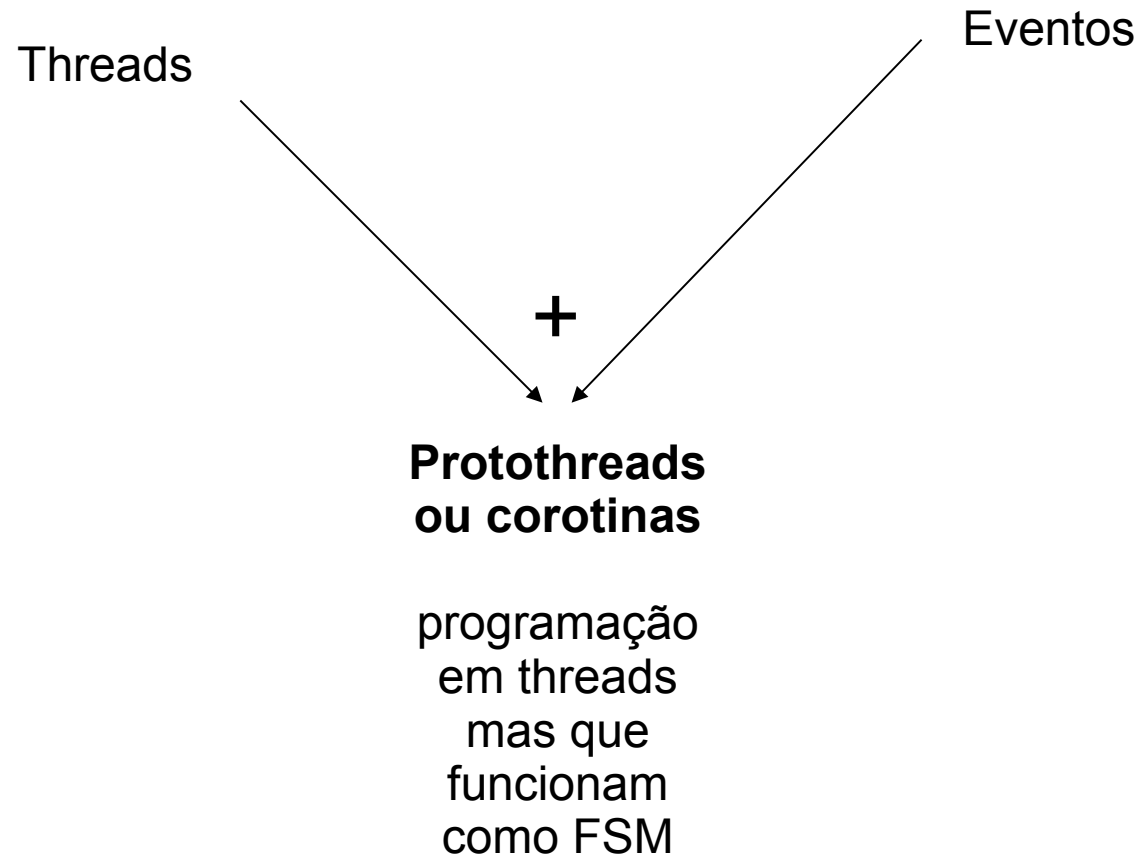
Projeto com eventos/FSM

**Com N estados pode se ter
até 2N novas transições**



```
enum {  
    ON,  
    WAITING,  
    OFF  
} state;  
  
void radio_wake_eventhandler() {  
    switch(state) {  
  
        case OFF:  
            if(timer_expired(&timer)) {  
                radio_on();  
                state = ON;  
                timer_set(&timer, T_AWAKE);  
            }  
            break;  
  
        case ON:  
            if(timer_expired(&timer)) {  
                timer_set(&timer, T_SLEEP);  
                if(!communication_complete()) {  
                    state = WAITING;  
                } else {  
                    radio_off();  
                    state = OFF;  
                }  
            }  
            break;  
  
        case WAITING:  
            if(communication_complete()  
                || timer_expired(&timer)) {  
                state = ON;  
                timer_set(&timer, T_AWAKE);  
            } else {  
                radio_off();  
                state = OFF;  
            }  
            break;  
    }  
}
```

Threads x eventos



Protothreads/corotinas

Idéia: seria possível fazer uma função que continuasse a partir do último ponto de saída? Isto é, que tivesse uma **continuação local**?

```
int funcao(void)
{
    int i;
    for (i = 0; i < 10; i++)
        return i; /* ponto de saída função */
}
```

Exemplo: seria possível fazer a função acima retornar a cada chamada o valor seguinte calculado no laço ? Assim...

```
X= funcao(); // X vale 0
X= funcao(); // X vale 1
X= funcao(); // X vale 2 ...
```

A isto se chama “continuação local” !

Protothreads/corotinas

Implementação de **continuação local** (usando goto's):

```
int funcao(void) {  
    static int i, state = 0;  
    switch (state) {  
        case 0: goto LABEL0;  
        case 1: goto LABEL1;  
    }  
    LABEL0: /* início da função */  
    for (i = 0; i < 10; i++) {  
        state = 1; /* assim vai voltar no LABEL1 */  
        return i;  
        LABEL1:; /* retorna logo após o ponto de saída */  
    }  
}
```

Funciona, mas é **difícil de ler e de manter** se tiver mais pontos de saída na função, pois necessita-se acrescentar um novo rótulo (LABEL) e uma nova entrada no *switch* a cada novo ponto de saída.

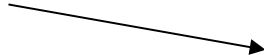
Ficaria melhor usar diretamente o *switch*, evitando os *goto's* !!!

Projeto de Sistemas Embarcados

Protothreads/corotinas

Implementação de **continuação local** (usando switch-case):

```
int funcao(void) {  
    static int i, state = 0;  
    switch (state) {  
        case 0:      /* início da função */  
        for (i = 0; i < 10; i++) {  
            state = 1; /* assim vai voltar no "case 1" */  
            return i;  
        case 1:; /* retorna logo após o ponto de saída */  
        }  
    }  
}
```



Note que o “case 1” está dentro do “for”

A linguagem C permite **switch-case aninhado com laços !!!**

Evita-se os gotos. Além disso é possível tornar a geração dos cases automática usando o préprocessador C.

Protothreads

Macros para **continuação local com switch-case**

```
struct pt { unsigned short lc; };

#define PT_INIT(pt)  (pt)->lc = 0

#define PT_BEGIN(pt)  switch((pt)->lc) { case 0:

#define PT_WAIT_UNTIL(pt, c)  (pt)->lc = __LINE__; case __LINE__: \
                                if(!(c)) return 1

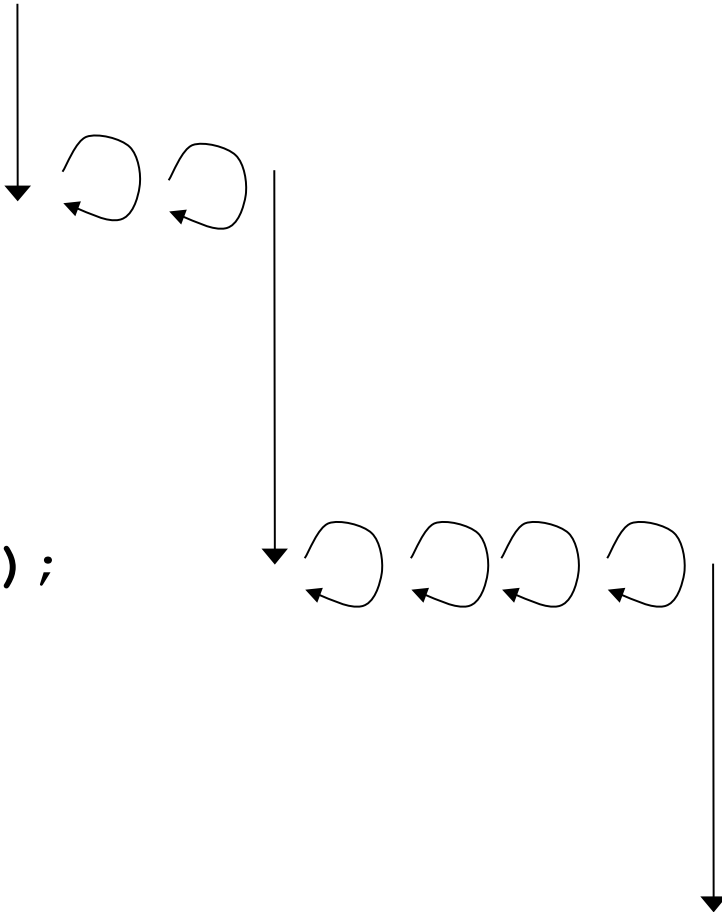
#define PT_EXIT(pt)  (pt)->lc = 0; return 0

#define PT_END(pt)  } (pt)->lc = 0; return 0
```

Fonte: biblioteca de protothreads (pt.h) –
dunkels.com/adam/download/pt-1.4.tar.gz

Um exemplo protothread

```
int a_protothread(struct pt *pt) {  
    PT_BEGIN(pt);  
    /* ... */  
    PT_WAIT_UNTIL(pt, condition1);  
    /* ... */  
    if(something) {  
        /* ... */  
        PT_WAIT_UNTIL(pt, condition2);  
        /* ... */  
    }  
    PT_END(pt);  
}
```



Protothread - expansão

```
int a_protothread(struct pt *pt) {  
    PT_BEGIN(pt);  
  
    PT_WAIT_UNTIL(pt, condition1);  
  
    if(something) {  
  
        PT_WAIT_UNTIL(pt, condition2);  
  
    }  
  
    PT_END(pt);  
}
```

```
int a_protothread(struct pt *pt) {  
    switch(pt->lc) { case 0:  
  
        pt->lc = 5; case 5:  
            if(!condition1) return 0;  
  
            if(something) {  
  
                pt->lc = 10; case 10:  
                    if(!condition2) return 0;  
  
            }  
  
        } return 1;  
    }
```

Protothread - expansão

```
int a_protothread(struct pt *pt) {  
    PT_BEGIN(pt);  
  
    PT_WAIT_UNTIL(pt, condition1);  
  
    if(something) {  
  
        PT_WAIT_UNTIL(pt, condition2);  
  
    }  
  
    PT_END(pt);  
}
```

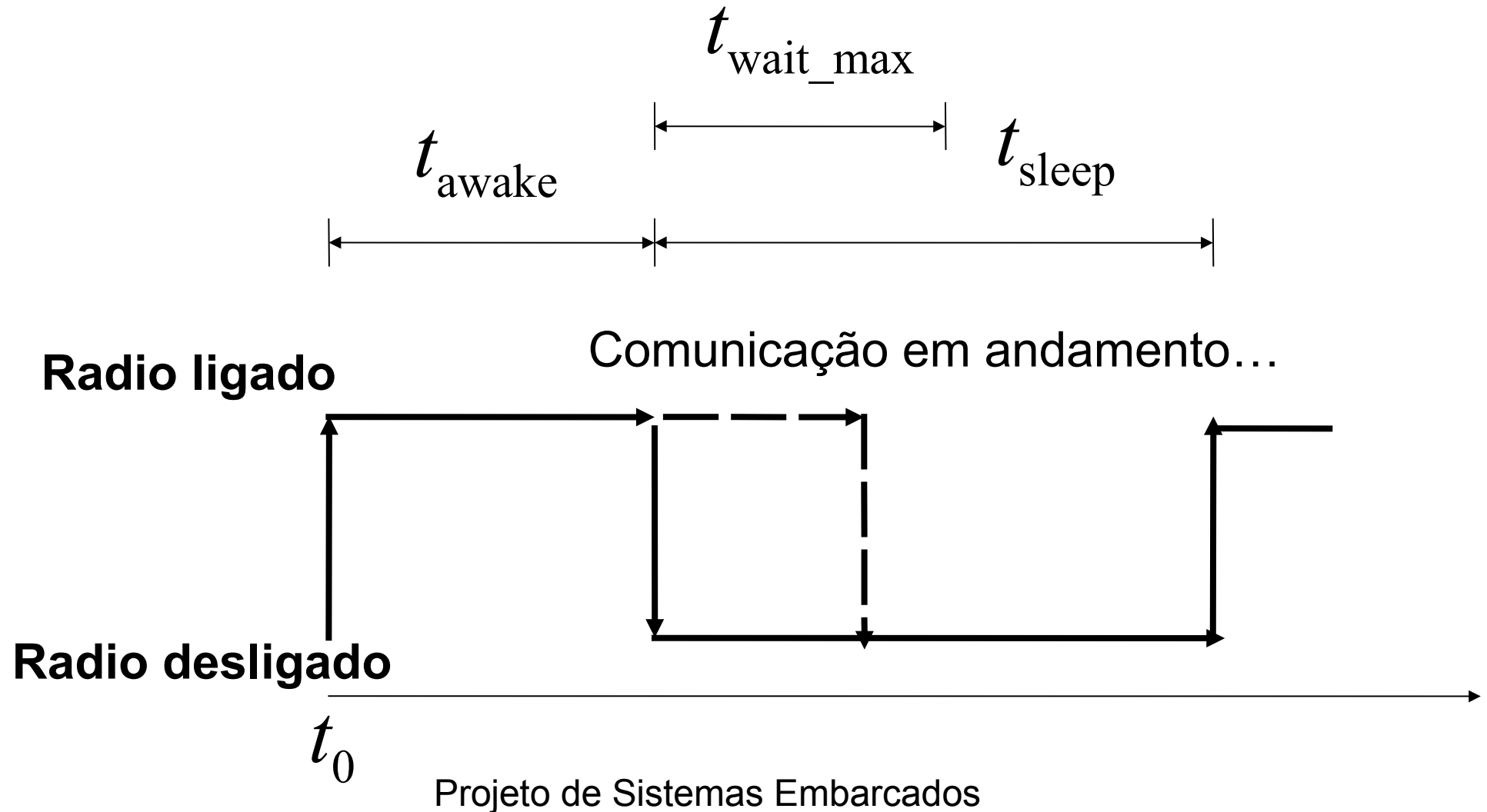
```
int a_protothread(struct pt *pt) {  
    switch(pt->lc) { case 0:  
  
        pt->lc = 5; case 5:  
        if(!condition1) return 0;  
  
        if(something) {  
  
            pt->lc = 10; case 10:  
            if(!condition2) return 0;  
  
        }  
  
    } return 1;  
}
```

Número das linhas

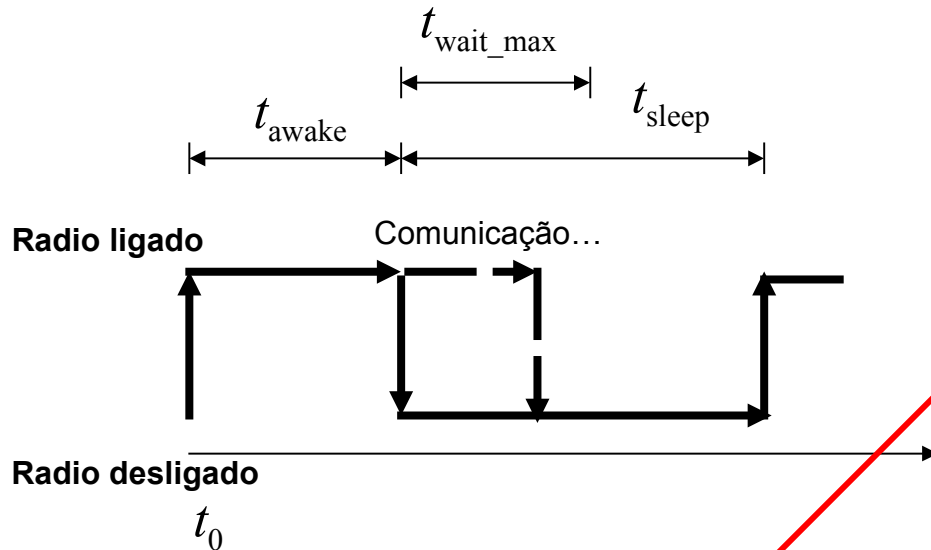
	Eventos/FSM	Threads	Protothreads
Lógica, controle de fluxo	Precisa ser organizado como máquina de estados. Pode ser mais difícil de entender a lógica.	Usa programação estruturada , cada thread é um programa independente* dos demais.	Programação estruturada , menos linhas de código do que FSM. Mas não tem como bloquear dentro de função (<i>thread pode!</i>)
Manutenção	Mais difícil de manter conforme mais estados são acrescentados	Manutenção mais fácil (mas depende das interdependências).	Mais fácil (como threads), mas menos problemas de interdependência.
Memória	Usa um única pilha, tende a usar menos memória RAM .	Cada thread tem uma pilha exclusiva. Usa mais RAM .	Usa um única pilha (como FSM), tende a usar menos memória RAM .
Depuração	Mais difícil . Necessário acompanhar transições de estados.	Mais fácil . Retém na pilha a sequência de chamadas de funções.	Mais fácil . Retém na pilha a sequência de chamadas de funções.
Testes	Pode se garantir o funcionamento correto , testando todas as sequências de eventos/estados.	Interdependências entre threads podem causar "bugs" de difícil detecção e correção .	Intermediário . Menos problemas de interdependência, pois não tem preempção.
Preempção e contexto.	Não possui . Contexto não é preservado.	Sim . Pode realizar preempção e manter o contexto.	Não possui . Contexto não é preservado.
Tempo real	Difícil de projetar.	Mais fácil de projetar. Técnicas de escalonamento bem conhecidas.	Intermediário . Técnicas de escalonamento estão sendo desenvolvidas.

Exemplo de uso: FSM convertida em protothread

Problema-exemplo: sistema de comunicação com rádio com consumo de energia reduzido



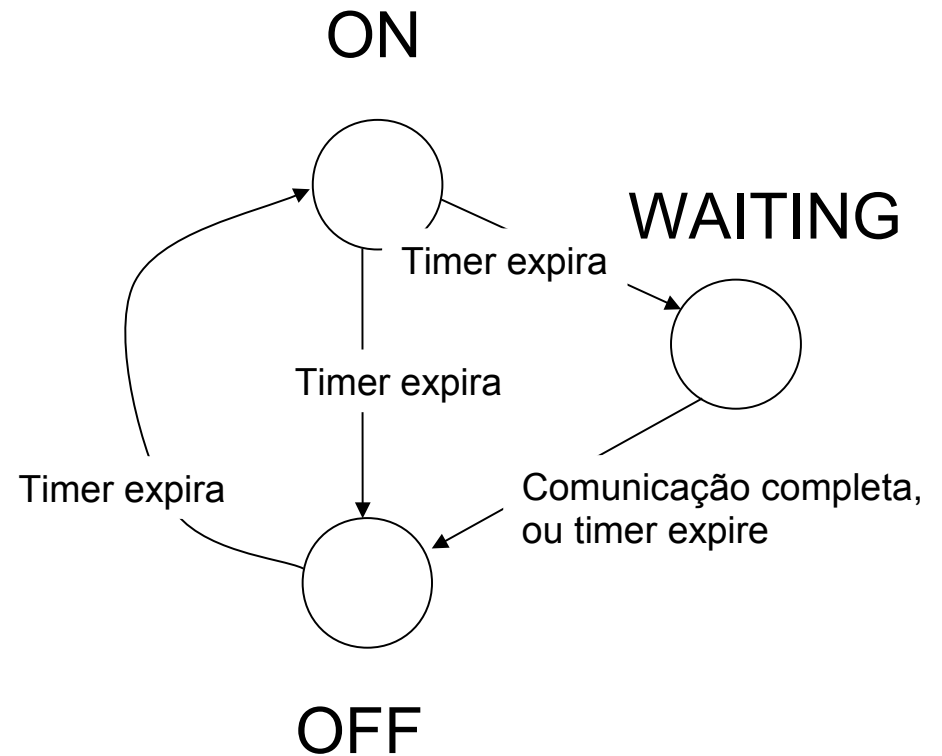
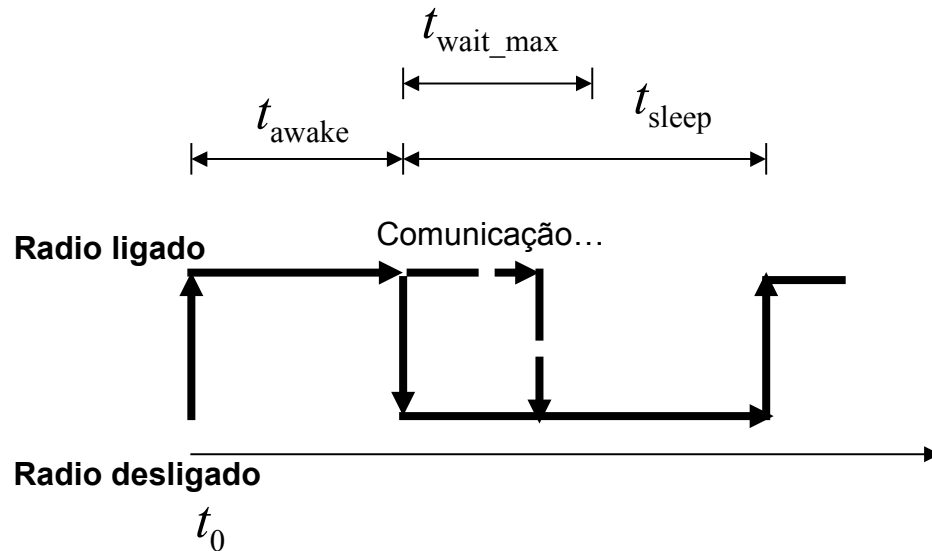
Algoritmo em 5 passos



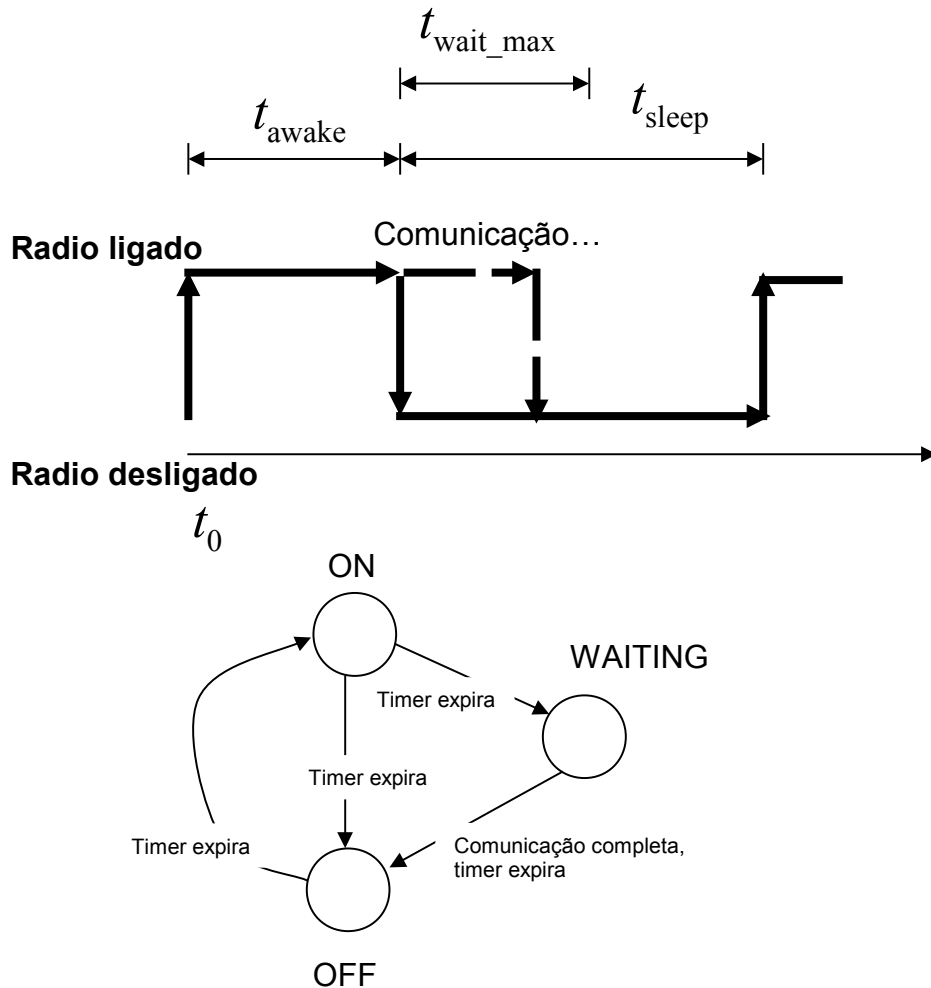
1. Ligar o rádio.
2. Esperar até $t = t_0 + t_{awake}$.
3. Se comunicação não completou, aguardar até completar ou $t = t_0 + t_{awake} + t_{wait_max}$.
4. Desligar rádio. Aguardar até $t = t_0 + t_{awake} + t_{sleep}$.
5. Repeat from step 1.

Precisa aguardar
neste ponto do código!

Projeto com FSM



Projeto com FSM

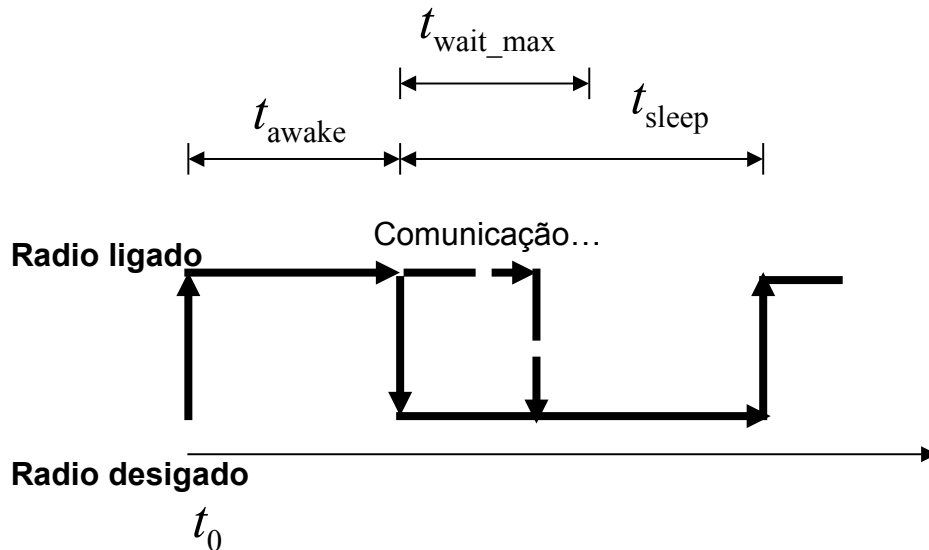


```

enum {ON, WAITING, OFF} state;

void eventhandler() {
    if(state == ON) {
        if(expired(timer)) {
            timer = t_sleep;
            if(!comm_complete()) {
                state = WAITING;
                wait_timer = t_wait_max;
            } else {
                radio_off();
                state = OFF;
            }
        }
    } else if(state == WAITING) {
        if(comm_complete() ||
           expired(wait_timer)) {
            state = OFF;
            radio_off();
        }
    } else if(state == OFF) {
        if(expired(timer)) {
            radio_on();
            state = ON;
            timer = t_awake;
        }
    }
}
  
```

Usando protothreads

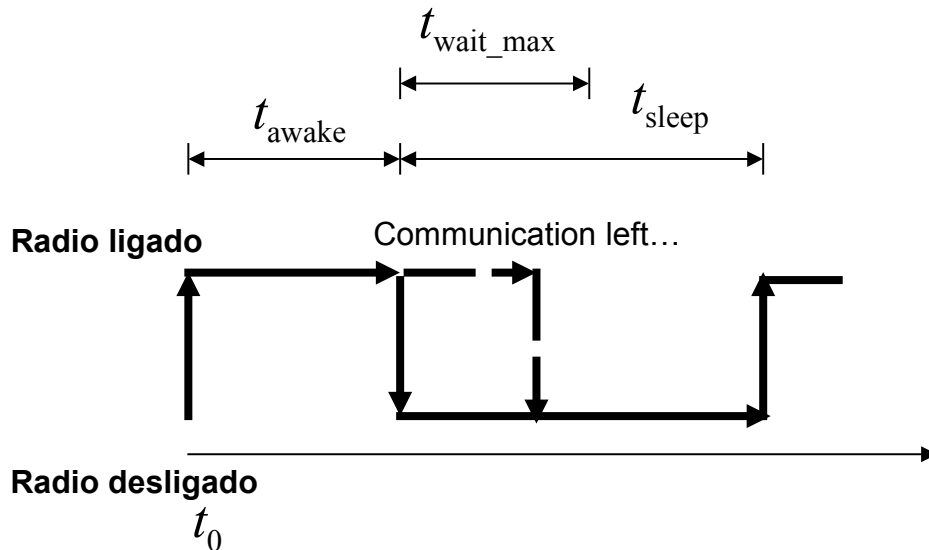


Protothreads –
permitem bloqueio
condicional:

`PT_WAIT_UNTIL()`

Fluxo de código é
sequencial, como uma
thread.

Protothreads



```
int protothread(struct pt *pt) {
    PT_BEGIN(pt);
    while(1) {
        radio_on();
        timer = t_aware;
        PT_WAIT_UNTIL(pt, expired(timer));
        timer = t_sleep;
        if(!comm_complete()) {
            wait_timer = t_wait_max;
            PT_WAIT_UNTIL(pt, comm_complete()
                          || expired(wait_timer));
        }
        radio_off();
        PT_WAIT_UNTIL(pt, expired(timer));
    }
    PT_END(pt);
}
```

- Código com menos linhas do que FSM.

Referências

Adam Dunkels - Protothreads - dunkels.com/adam/pt

Simon Tatham. Coroutines in C.
www.chiark.greenend.org.uk/~sgtatham/coroutines.html

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06). ACM

Dunkels, Adam, Oliver Schmidt, and Thiemo Voigt. "Using protothreads for sensor node programming." Proceedings of the REALWSN. Vol. 5. 2005.