

EGM0017 (60h)

Fluxo e metodologias de projeto de Sistemas Embarcados

Prof. Josenalde Barbosa de Oliveira – UFRN



josenalde.oliveira@ufrn.br

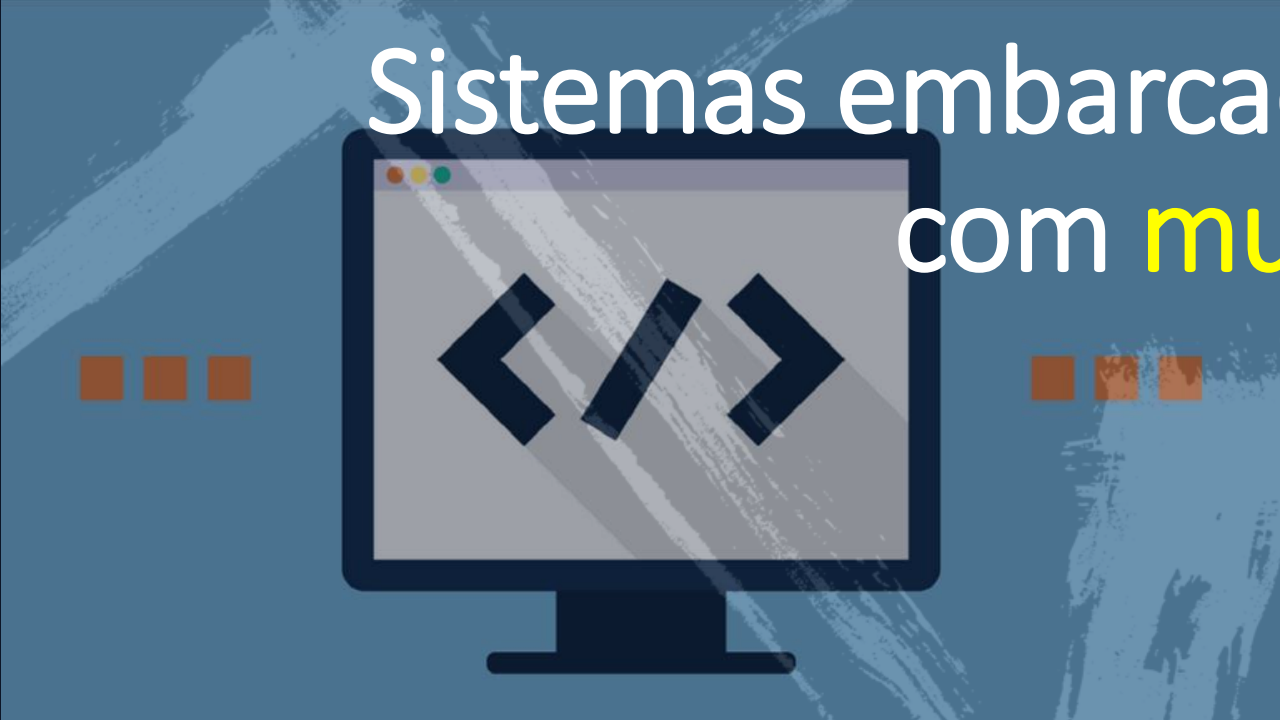
Programa de Pós-Graduação em Engenharia Mecatrônica



Concorrência e paralelismo – dividir para conquistar

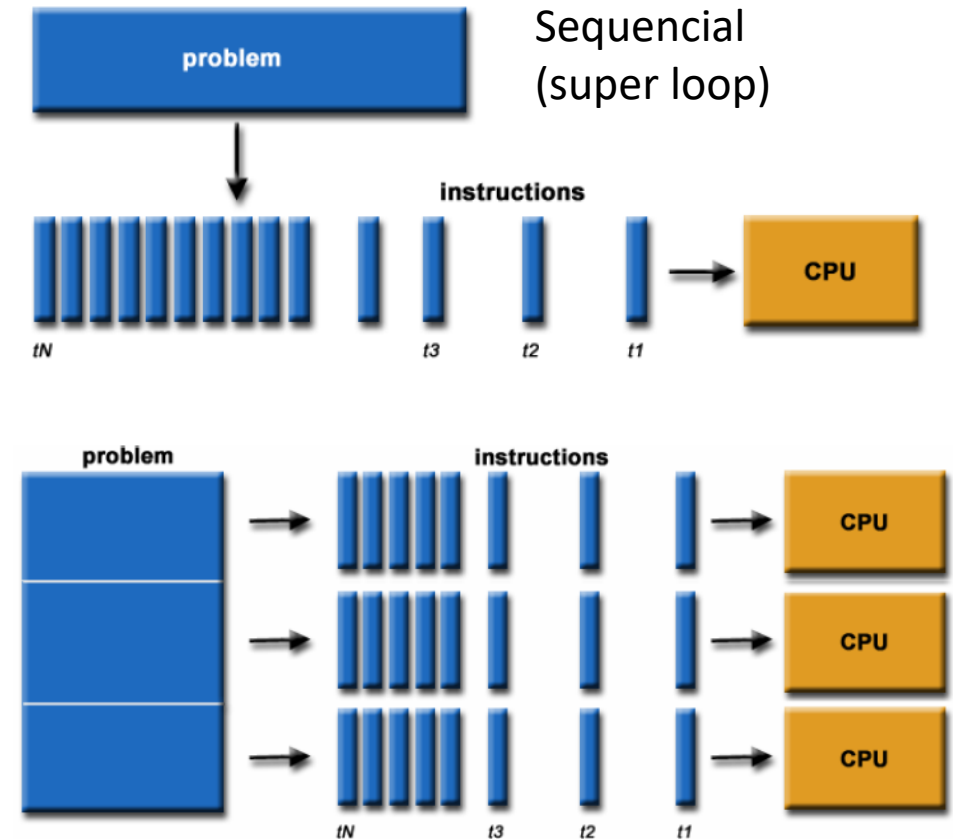


Sistemas embarcados de tempo real com multi-core



Agregando o paradigma paralelo+concorrente ao *toolbox dev para sistemas embarcados com RTOS*

- Reduzir o tempo para solucionar um problema
- *Resolver problemas mais complexos E de maior dimensão*
- Códigos melhor gerenciáveis (manutenção)
- Potencializa tolerância a falhas atendendo a requisitos não funcionais de redundância de hardware, software, tempo
- **Objetivo:** garantir a execução de funções corretamente em intervalo $[0,t]$, assumindo que está funcionando corretamente no tempo 0 (início)
- Há contudo um trade-off, pois pode aumentar consumo, temperatura, tempo de modelagem e implementação



Paralelo: instruções sequenciais Tasks paralelas em cada nó de processamento, (RTOS) de forma **concorrente**
Introduz OVERHEAD

Concurrency

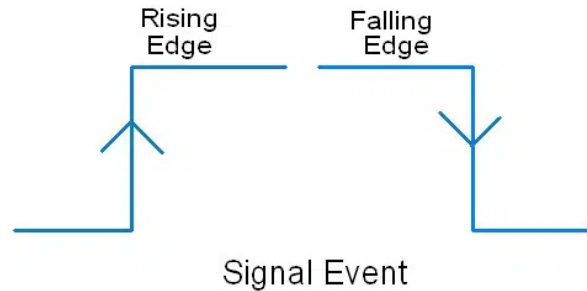
Concurrency happens whenever different parts of your program might execute at different times or out of order. In an embedded context, this includes:

- interrupt handlers, which run whenever the associated interrupt happens,
- various forms of multithreading, where your microprocessor regularly swaps between parts of your program,
- and in some systems, multiple-core microprocessors, where each core can be independently running a different part of your program at the same time.

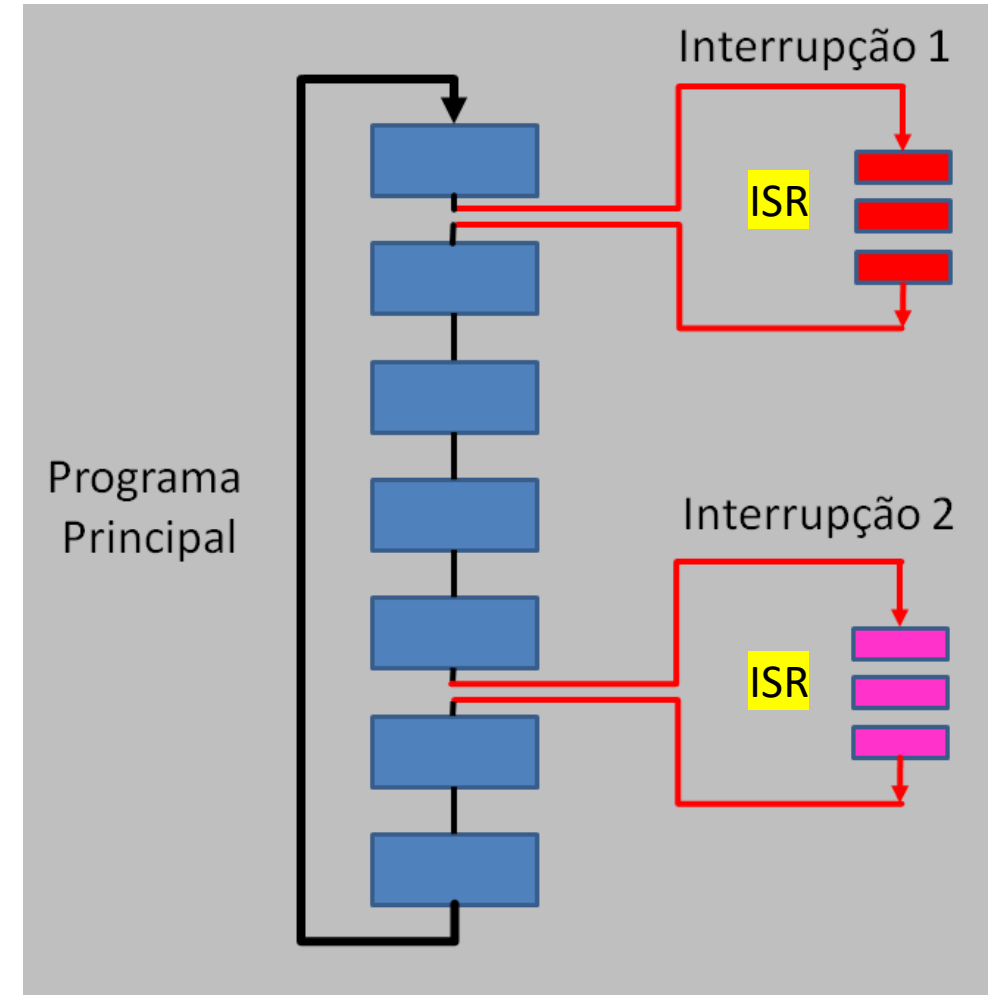
Since many embedded programs need to deal with interrupts, concurrency will usually come up sooner or later, and it's also where many subtle and difficult bugs can occur. Luckily, Rust provides a number of abstractions and safety guarantees to help us write correct code.

Interrupção de hardware no ESP32

- *Leitura polling*: varredura em todo GPIO por mudança de estado
- *Leitura por interrupção*: apenas GPIO de interesse
- Modos de detecção:



- **FALLING**: um modo que faz ser gerada uma interrupção quando um GPIO vai do nível alto (3V3) para nível baixo (0V). Ou seja, interrupção gerada na transição de nível alto para baixo;
- **RISING**: um modo que faz ser gerada uma interrupção quando um GPIO vai do nível baixo (0V) para nível alto (3V3). Ou seja, interrupção gerada na transição de nível baixo para alto;
- **LOW**: um modo que faz ser gerada uma interrupção gerada quando o GPIO está em nível baixo;
- **HIGH**: um modo que faz ser gerada uma interrupção gerada quando o GPIO está em nível alto;
- **CHANGE**: um modo que faz ser gerada uma interrupção quando há qualquer transição de nível no GPIO. Ou seja, tanto de nível baixo para alto quanto de nível alto para baixo.



micros() works initially but will start behaving erratically after 1-2 ms. delayMicroseconds() does not use any counter, so it will work as normal.

Interrupção de hardware no ESP32

<https://reference.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/>

- *Interruption Service Routine (ISR)*

- Não recebe parâmetro nem retorna valores
 - comunicação entre main() e ISR com variáveis globais voláteis de acesso controlado
- O mais rápido possível
- Com múltiplos ISRs, apenas 1 por vez
- Funções de espera bloqueantes *delay()* usam interrupções, não devem ser usadas dentro de ISR
- Obtenção de tempo desde o início da carga do firmware (upload) não incrementa com *millis()* na ISR
- micros() pode funcionar erráticamente após 1-2ms (recomenda-se até 500us +/-) – ver resolução de acordo com clock e PLL
- delayMicroseconds() não usa contadores/interrupção
- Por recomendação geral não usar métodos bloqueantes ou de temporização em ISR

Esp-idf

```
unsigned long IRAM_ATTR micros() {  
    return (unsigned long)(esp_timer_get_time());  
}  
void IRAM_ATTR delayMicros(uint32_t us) {  
    uint32_t m = micros();  
    if (us) {  
        uint32_t e = (m + us);  
        if (m > e) { //overflow  
            while (micros() > e) { NOP(); }  
        }  
        while (micros() < e) { NOP(); }  
    }  
}
```

Abordagem bare-metal tratar região crítica

de variável alterada em ISR com noInterrupts() e interrupts()

```
noInterrupts ();  
    long myCounter = isrCounter;  
interrupts ();
```


Interrupção de hardware no ESP32

DOIT ESP32 DEVKIT V1

PINOUT

Chip-enable signal, Active High.				EN	pin15				
ADC_PA	RTC_GPIO0	ADC1_CH0	SENSOR_VP	GPI036	pin14				
ADC_PA	RTC_GPIO3	ADC1_CH3	SENSOR_VN	GPI039	pin13				
	RTC_GPIO4	ADC1_CH6	VDET1	GPI034	pin12				
	RTC_GPIO5	ADC1_CH7	VDET2	GPI035	pin11				
XTAL_32kHz	Touch9	RTC_GPIO9	ADC1_CH4	GPI032	pin10				
XTAL_32kHz	Touch8	RTC_GPIO8	ADC1_CH5	GPI033	pin9				
	DAC_1	RTC_GPIO6	ADC2_CH8	EMAC_RXD0	GPI025	pin8			
	DAC_2	RTC_GPIO7	ADC2_CH9	EMAC_RXD1	GPI026	pin7			
	Touch7	RTC_GPIO17	ADC2_CH7	EMAC_RX_DV	GPI027	pin6			
HS2_CLK	SD_CLK	HSPI_CLK	MTMS	Touch6	RTC_GPIO16	ADC2_CH6	EMAC_TXD2	GPI014	pin5
HS2_DATA2	SD_DATA2	HSPI_MISO	MTDI	Touch5	RTC_GPIO15	ADC2_CH5	EMAC_TXD3	GPI012	pin4
HS2_DATA3	SD_DATA3	HSPI_MOSI	MTCK	Touch4	RTC_GPIO14	ADC2_CH4	EMAC_RX_ER	GPI013	pin3
				GND	pin2				
				VIN	pin1				

POWER

GND

Serial Pin

Analog Pin


Control

Physical Pin

Port Pin

Touch Pin

DAC Pin



The image shows a DOIT ESP32 DevKit V1 board. The central component is the ESP-WROOM-32 module, which has a gold-colored PCB and a black chip. The chip is labeled with 'ESP-WROOM-32', 'CE 1313', and 'FCC ID: 2AC7Z-ESPWROOM32'. The board has a black PCB with various components, including a USB Type-C port, a USB Type-A port, a 3.5mm audio jack, and a push button. The pins are labeled with their functions and pin numbers. The pinout is as follows:

pin15	GPI023	SPI_MOSI	HS1_STROBE						
pin14	GPI022	EMAC_TXD1	U0RTS	I2C_SCL					
pin13	GPI01	EMAC_RXD2	U0TXD	CLK_OUT3					
pin12	GPI03		U0RXD	CLK_OUT2					
pin11	GPI021	EMAC_TX_EN		I2C_SDA					
pin10	GPI019	EMAC_TXD0	U0CTS	SPI_MISO					
pin9	GPI018		SPI_CLK	HS1_DATA7					
pin8	GPI05	EMAC_RX_CLK	SPI_CS0	HS1_DATA6					
pin7	GPI017	EMAC_CLKOUT180	U2_TXD	HS1_DATA5					
pin6	GPI016	EMAC_CLKOUT	U2_RXD	HS1_DATA4					
pin5	GPI04	EMAC_TX_ER	ADC2_CH0	RTCIO10	Touch0	HSPIHD	SD_DATA1	HS2_DATA1	
pin4	GPI02		ADC2_CH2	RTCIO12	Touch2	HSPIWP			
pin3	GPI015	EMAC_RXD3	ADC2_CH3	RTCIO13	Touch3	MTD0	HSPI_CS0	SD_CMD	HS2_CMD
pin2	GND								
pin1	VDD_3V3								

Interrupção de hardware no ESP32

Exemplo: um código que conta apertos no botão... (configuração pull-down)

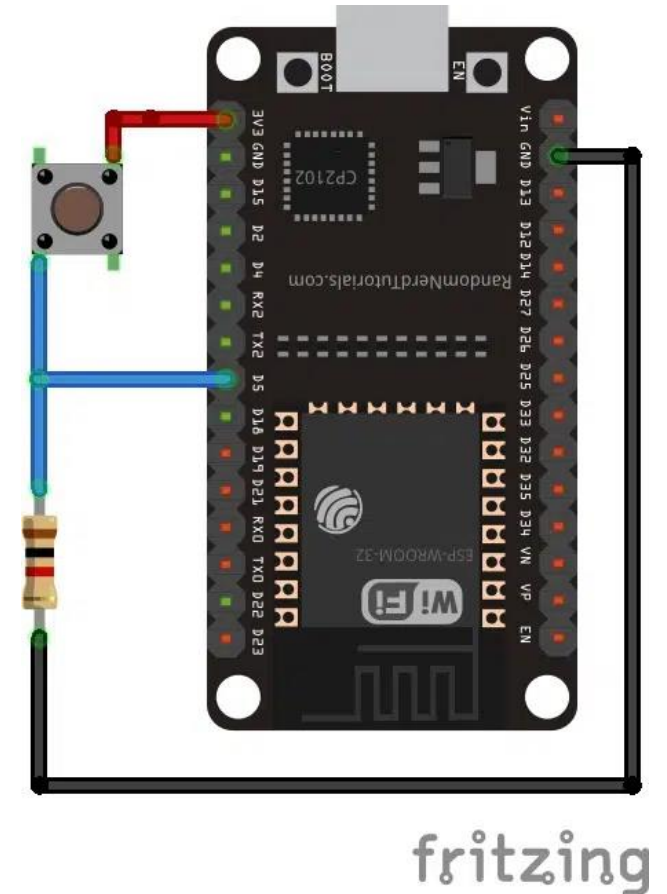
```
#define GPIO_BOTAO 5

volatile int contador_acionamentos = 0;

/* Função ISR (chamada quando há geração da interrupção) */
void IRAM_ATTR contaPressionamentos() {
/* Conta acionamentos do botão considerando debounce tratado */
    contador_acionamentos++;
}

void setup() {
    Serial.begin(115200);
/* Configura o GPIO do botão como entrada e configura interrupção externa no modo RISING para ele. */
    pinMode(GPIO_BOTAO, INPUT); // INPUT_PULLUP dispensa resistor
    attachInterrupt(GPIO_BOTAO, contaPressionamentos, RISING);
}

void loop() { //CPU executando continuamente
    Serial.print("Acionamentos do botao: ");
    Serial.println(contador_acionamentos);
    delay(1000);
}
```



<https://www.makerhero.com/blog/uso-de-interruptoes-externas-com-esp32/>

Paralelismo implícito e explícito

Implícito: determinado pelo compilador, o qual infere regiões de paralelismo, atribui tarefas em paralelo, controla e sincroniza execução/comunicação (Exemplos: *g++ -fopenmp -fparallelize-loops=N, #pragma omp parallel, API stream() Java, joblib Python*)

- Mais fácil para o programador, porém sem garantia de eficiência a depender do problema

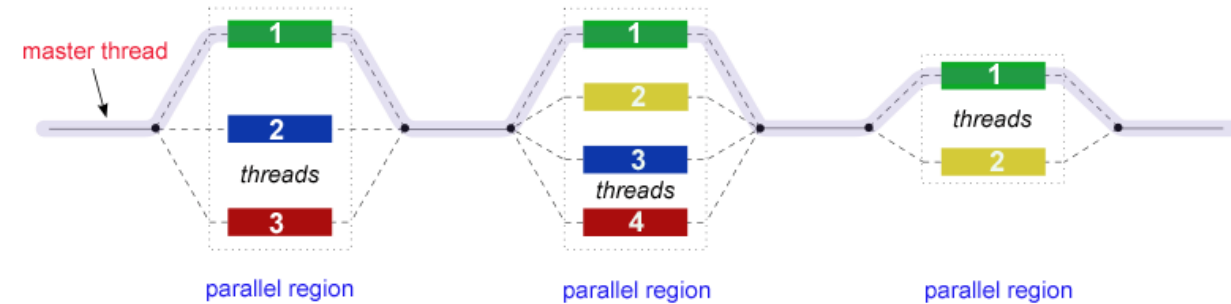
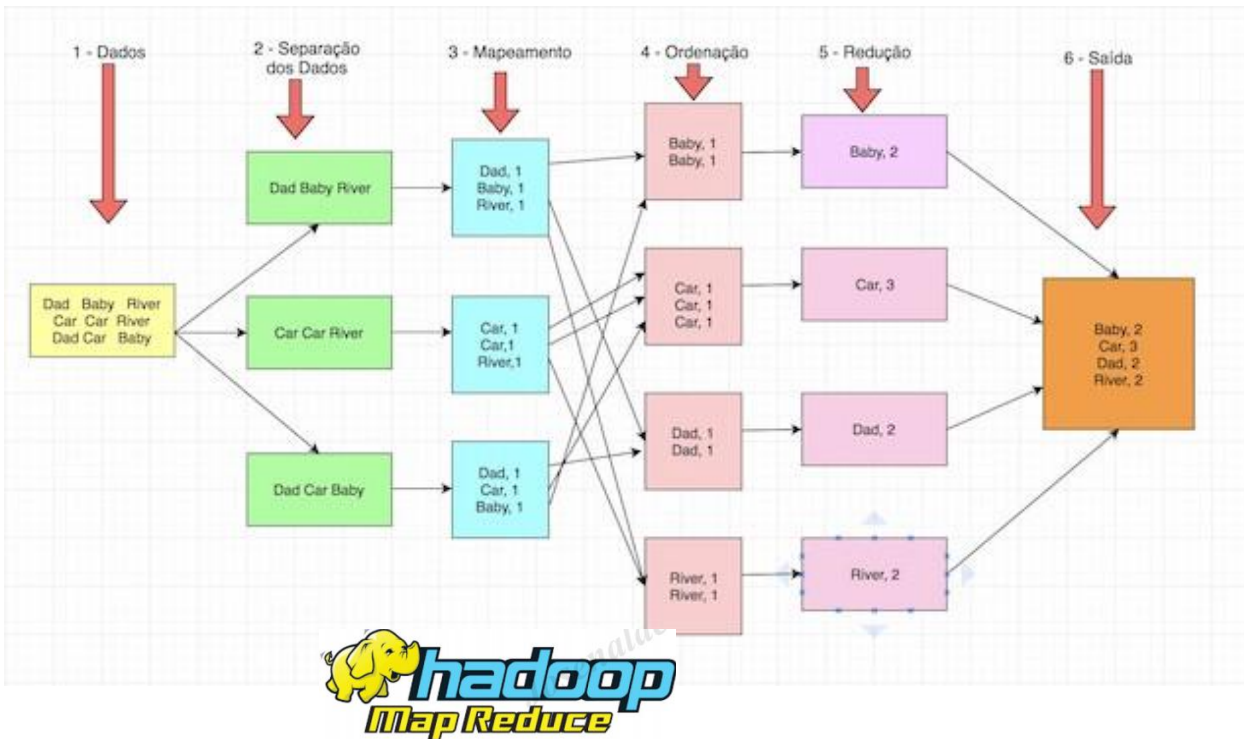
Explícito: cabe ao programador definir tarefas para execução em paralelo, distribuição de processadores, inserir pontos de sincronização (Exemplos: *#pragma c/c++, multiprocessing python, Java threads, Workers JS, MPI etc.*)

Aspectos sobre programação paralela

Ideia principal: dividir problema complexo em partes mais simples

- *computação numérica: multiplicação matricial*
- *manipulação de dados: mapreduce e spark: MESMA IDEIA DO FORK - JOIN*

Dificuldades: *algoritmos, integração, codificação*

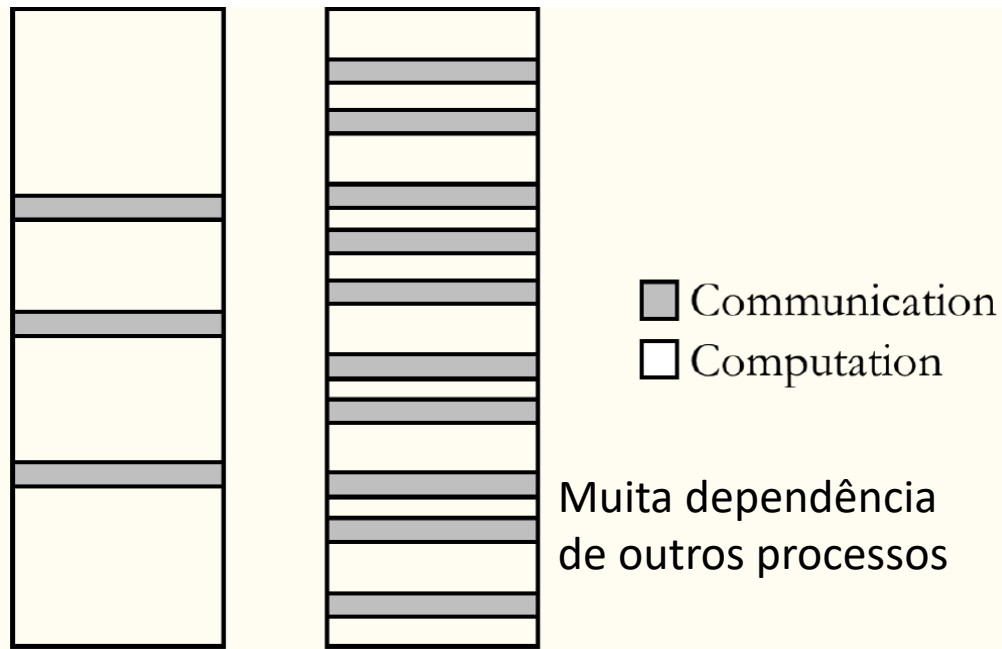


Aspectos sobre programação paralela

Escalabilidade

- **altamente**/fracamente: Problemas que são facilmente paralelizáveis

Granularidade: *coarse (grossa): mais computação que comunicação; fine (fina): mais comunicação*



Pouca dependência
de outros processos

Bottlenecks:

- Dependência entre dados (flow dependence:fd)
 - Um processo aguardando outro
 - T1 lê posição na mem após T2 gravar

```
1.sum=a+1
2.first_term=sum*scale1
-----
3.sum=b+1
4.second_term=sum*scale2
```

```
1.first_sum=a+1
2.first_term=first_sum*scale1
-----
3.second_sum=b+1
4.second_term=second_sum*scale2
```

fd(1,2); fd(3,4) – independentes?

Se paralelizar, 2 e 3 anti-dependência (restringe concorrência)

Solução possível

Exemplo de altamente escalável: somatório

Seja um array **x** de tamanho **n** = {6, 4, 16, 10, 12, 14, 2, 8}

```
//o básico
sum = 0
for (i=0; i<n; i++) {
    sum += x[i];
}
```

Não há ordem definida
Associa e acumula

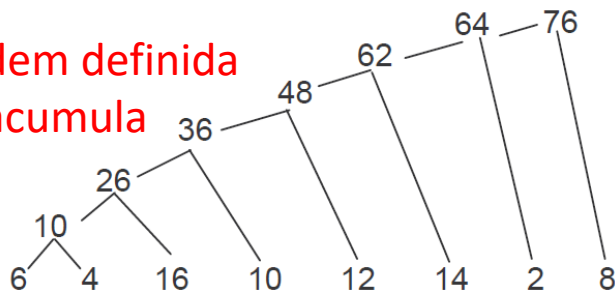


Figure 1.1. Summing in sequence. The order of combining a sequence of numbers (6, 4, 16, 10, 12, 14, 2, 8) when adding them to an accumulation variable.

Forma 'comum' de pensar

Outra forma de 'ver' com pares de somas (par/ímpar):

$(x_0 + x_1) + (x_2 + x_3), (x_4 + x_5) + (x_6 + x_7), \dots$

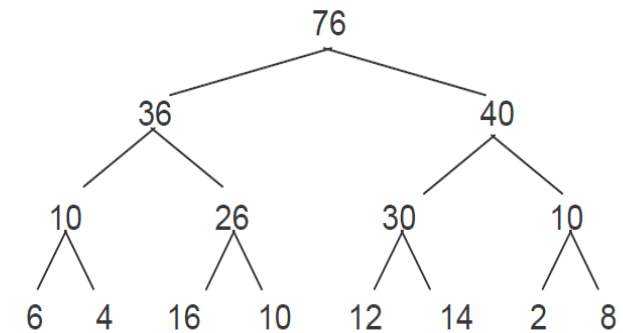


Figure 1.2. Summing in pairs. The order of combining a sequence of numbers (6, 4, 16, 10, 12, 14, 2, 8) by (recursively) combining pairs of values, then pairs of results, etc.

1 processador: 7 somas na solução 1 e na solução 2!
P = N/2 processadores. Somas de mesmo nível em paralelo, sem dependências.

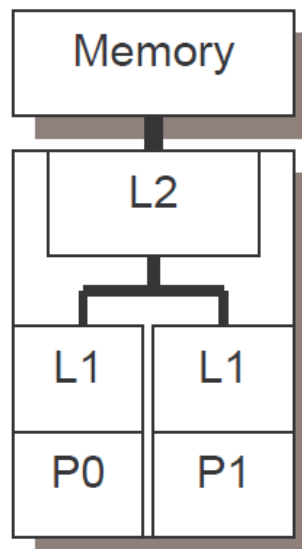
Exemplo: contar items numa lista

```
int *array;
int length;
int count;

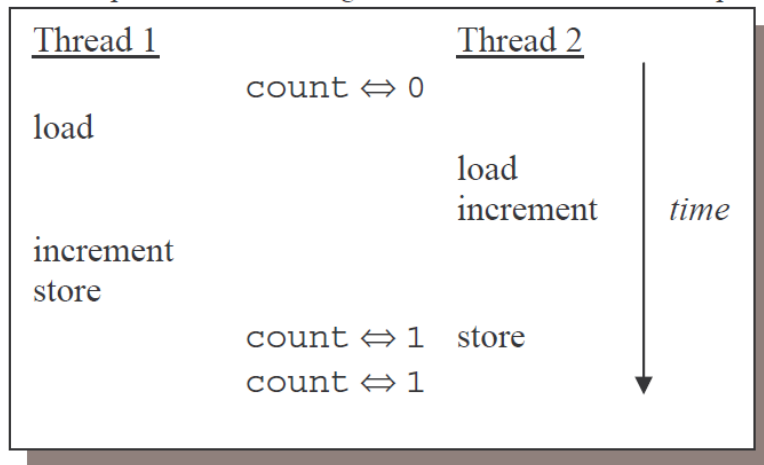
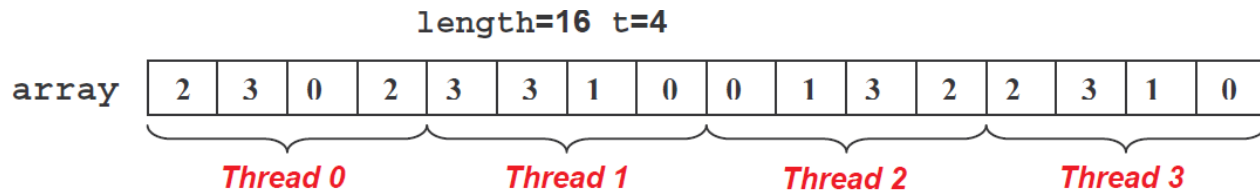
int count3s() {
    int i;
    count = 0;
    for (i=0; i<length; i++) {
        if(array[i]==3) count++;
    }
    return count;
}
```

Solução 1: sequencial

https://github.com/josenalde/flux-embedded-design/blob/main/src/count3s_s.cpp



Exemplo: contar items numa lista



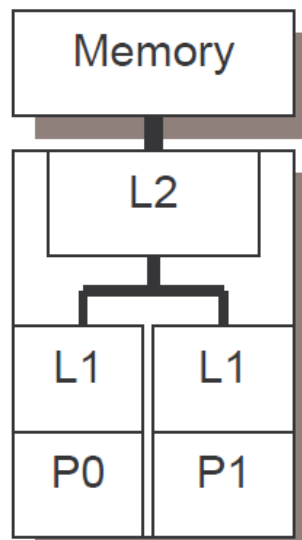
Recurso compartilhado pode gerar 'bug'
Race condition: **condição de corrida**

```

1 int t;          /* number of threads */
2 int *array;
3 int length;
4 int count;
5
6 void count3s ()
7 {
8     int i;
9     count = 0;
10    /* Create t threads */
11    for (i=0; i<t; i++)
12    {
13        thread_create (count3s_thread, i);
14    }
15
16    return count;
17 }
18
19 void count3s_thread (int id)
20 {
21     /* Compute portion of the array that this thread should work on */
22     int length_per_thread = length/t;
23     int start = id * length_per_thread;
24
25     for (i=start; i<start+length_per_thread; i++)
26     {
27         if (array[i] == 3)
28         {
29             count++;
30         }
31     }
32 }

```

Solução 2: multithread



https://github.com/josenalde/flux-embedded-design/blob/main/src/pthread_count3s_1.cpp

Mais 'gente' trabalhando...precisam de um BOM gestor/escalonador

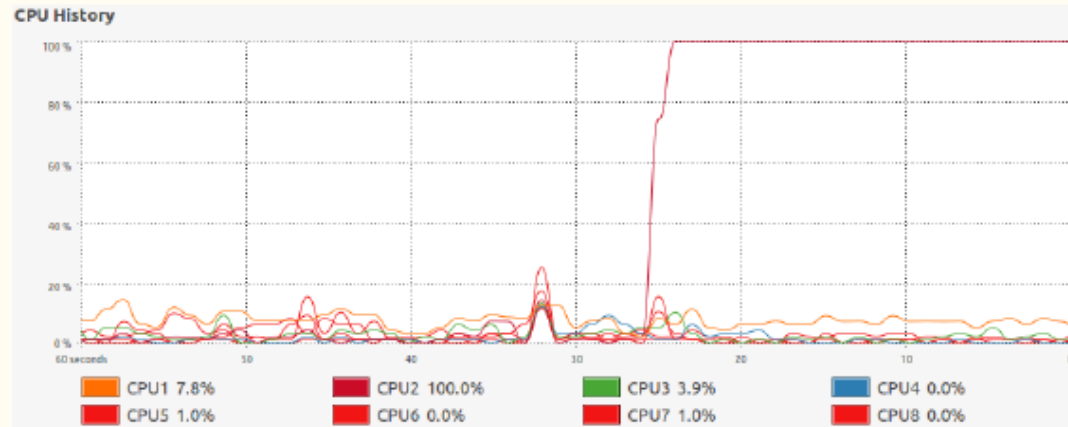
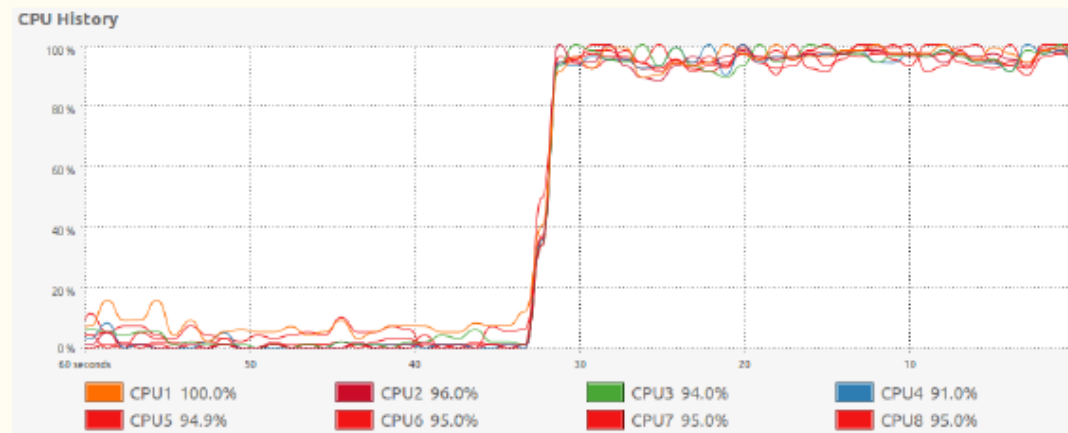


FIGURE 2: 8 Cores Used for Computation

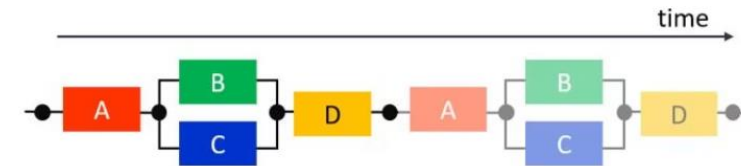


Sobre threads POSIX – embedded Linux, Windows GPOS, EGPSO



```
#include <pthread.h>
int errCode;
```

```
int main() {
    pthread_t tid[MAX]; // array de threads
    /*
        parte sequencial
    */
    // começa a paralelizar
    for(int i = 0; i < MAX; i++) {
        errCode = pthread_create(&tid[i], NULL, <nome_função_a_processar>, (void *) i);
        // se errCode == 0 SUCESSO na criação. Outros códigos em <errno.h>
    }
    // finaliza paralelização juntando resultados
    for (int i = 0; i < MAX; i++) {
        errCode = pthread_join(tid[i], NULL);
    }
}
```



A assinatura da função a processar precisa ter retorno void* e parâmetro void*

Exemplo: void* test(void* num); dentro da função faz **typecast** para o tipo do dado desejado

Sobre threads POSIX

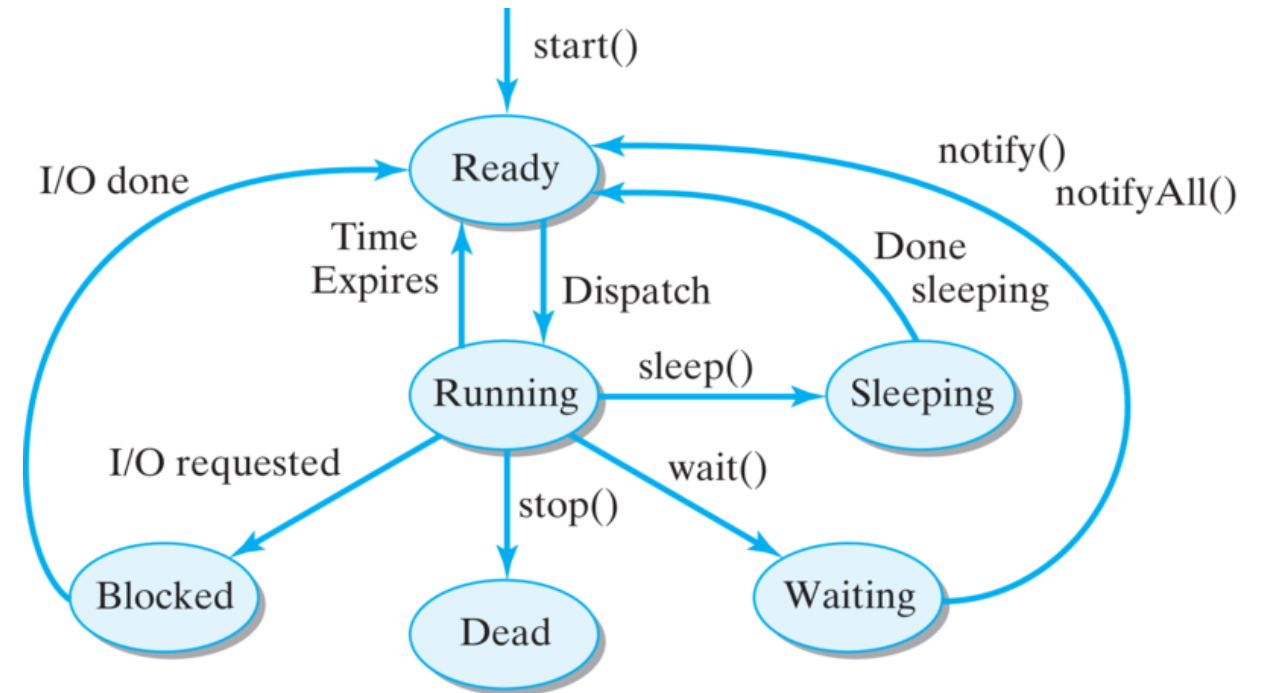
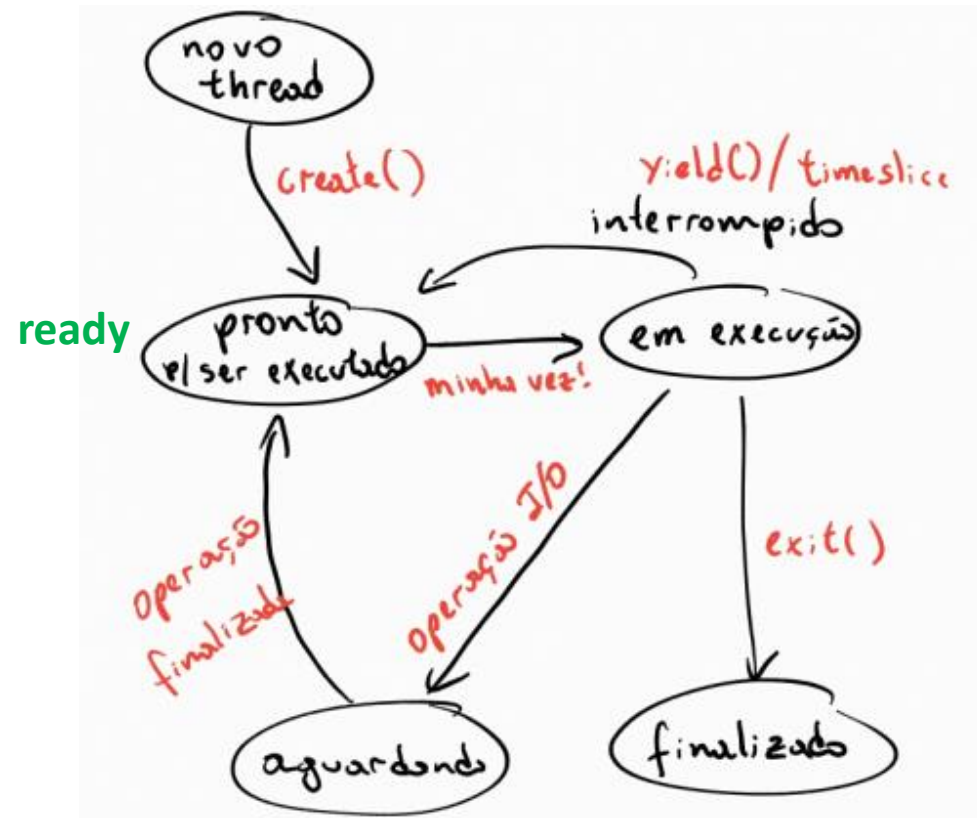


```
#include <pthread.h>
int errCode;

int main() {
    pthread_t tid[MAX]; // array de threads
    /*
        parte sequencial
    */
    // começa a paralelizar
    for(int i = 0; i < MAX; i++) {
        errCode = pthread_create(&tid[i], NULL, <nome_função_a_processar>, (void *) i);
        // se errCode == 0 SUCESSO na criação. Outros códigos em <errno.h>
    }

    int pthread_create(
        pthread_t *tid,
        const pthread_attr_t *attr, //atributos da thread (pode ser NULL)
        void *(*start_routine)(void *) //ponteiro para função a executar,
        void *arg
    );
}
```

Sobre threads POSIX – estados da thread



Sobre threads POSIX – atributos na criação



```
for(int i = 0; i < MAX; i++) {  
    errCode = pthread_create(&tid[i], NULL, <nome_função_a_processar>, (void *) i);  
    // se errCode == 0 SUCESSO na criação. Outros códigos em <errno.h>  
}
```

```
int pthread_create(  
    pthread_t *tid,  
    const pthread_attr_t *attr, //atributos da thread (pode ser NULL)  
    void *(*start_routine)(void *) //ponteiro para função a executar,  
    void *arg  
);
```

```
pthread_attr_t myAttr;  
pthread_t myThread;  
  
pthread_attr_init(&myAttr);  
  
pthread_attr_setscope(&myAttr, PTHREAD_SCOPE_SYSTEM); //ou _PROCESS  
pthread_attr_setdetachstate(&myAttr, PTHREAD_CREATE_JOINABLE); //ou _DETACHED  
// escopo SYSTEM e tipo JOINABLE são default, por isso deixamos NULL  
pthread_create(&myThread, &myAttr, <func>, (void *) PARAM);
```

The thread competes for resources with all other threads in all processes on the system that are in the same scheduling allocation domain a group of one or more processors). **PTHREAD_SCOPE_SYSTEM** threads are scheduled relative to one another according to their scheduling policy and priority.

Sobre threads POSIX



```
// finaliza paralelização juntando resultados
for (int i = 0; i < MAX; i++) {
    errCode = pthread_join(tid[i], NULL);
}
//Uma vez a thread reunida, não mais existe,
//seu thread_id não é mais válido e não pode ser
//reunida a outra thread
```

```
int pthread_join(
    pthread_t tid,
    void **status //deve passar na chamada
                  //o endereço de um ponteiro
                  // status de saída (return_thread)
);
```

Poderia ser criado um ponteiro `void * thread_res` para ser usado no `_join`

```
//errCode = pthread_join(tid[i], &thread_res);
if errCode != 0 //ERRO NA FINALIZAÇÃO DA THREAD
else cout << "Thread finalizada com sucesso... com retorno: " << (char *)thread_res;
```

Para finalizar explicitamente uma thread `void pthread_exit (void *retval);`
ESTE VALOR EM `*retval` na saída da thread é associado ao ponteiro `*thread_res` de status no JOIN!

Para bloquear uma thread, permitindo que outra entre em execução:

```
int sched_yield (void)//depracted;
usleep(100); //por exemplo usleep, de unistd.h (no Windows Sleep)
```



Códigos para fixação

- https://github.com/josenalde/flux-embedded-design/blob/main/src/pthread_validade_simple.cpp
- https://github.com/josenalde/flux-embedded-design/blob/main/src/pthread_validade_simple_2.cpp